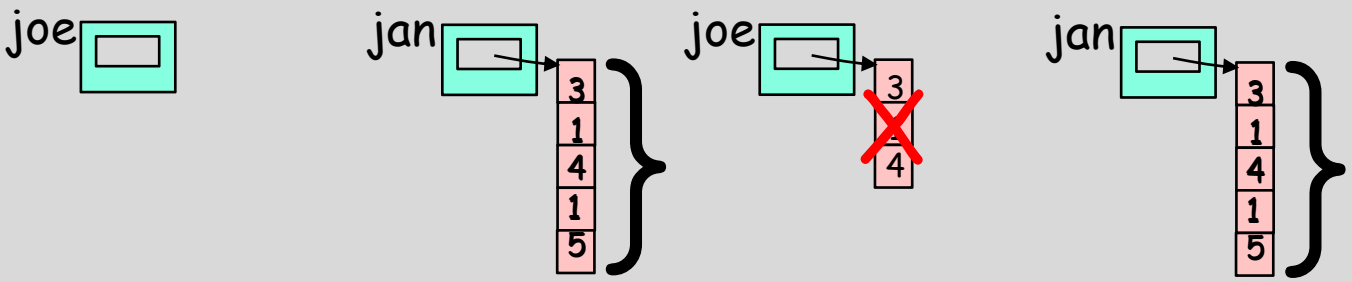
Lecture #4

- Resource Management, Part 2
  - Assignment Operators
- Basic Linked Lists
  - Insertion, deletion, destruction, traversals, etc.
- Advanced Linked Lists
  - Tail Pointers
  - Doubly-linked Lists
- Appendix: For on-your-own study (optional)
  - Linked Lists with Dummy Nodes

Assignment Operators…
What's the big picture?

Just as we need a special copy constructor function to create a new class variable from an existing one…



```
void constructJoeFromJan()
{
  PiNerd jan(5);

  PiNerd joe(jan);

  …
}
```

```
void reassignJoeToJan()
{
  PiNerd joe(3), jan(5);

  joe = jan;

  …
}
```

We often need to create a special assignment function to correctly change an existing variable's value to another existing variable. It's called automatically when we use =.

The Assignment Operator

```
int main()
{
    Circ  x(1,2,3);

    Circ  y = x; // #1
}
```

New variable

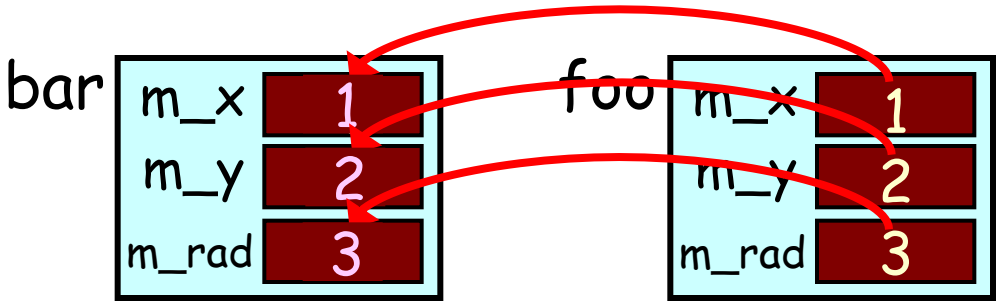
Existing variable

```
int main()
{
    Circ  foo(1,2,3);

    Circ  bar(4,5,6);

    bar = foo;  // #2
}
```

Existing variable

Existing variable

- When we create a new variable (y) from an existing one (x) as on line #1, C++ calls the copy constructor function (if one is defined)
- In contrast, if we change an existing variable's value (bar) by setting it to another existing variable (foo) as on line #2 C++ calls a different function called an assignment operator (if one exists) to copy the data
- If you don't write your own assignment operator function, then when you do an assignment like:
  bar = foo;
  C++ will "shallow copy" all of the data, byte for byte, to the new variable



The Assignment Operator

foo

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
     m_x = x; m_y = y; m_rad = r;
  }
  void setMeEqualTo(const Circ &src)
  {
    m_x = src.m_x;
    m_y = src.m_y;
    m_rad = src.m_rad;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

- Let's define a method called setMeEqualTo that sets the current Circle variable equal to the Circle variable that's passed in.
- The code below:
  bar.setMeEqual(foo);
  copies the parameter's (foo) values (x, y and rad) into the target variable (bar).
- This is essentially what a simple assignment operator does, but we've cheated on the syntax.

```
int main()
{
  Circ  foo(1,2,3);

  Circ  bar(4,5,6);

  bar.setMeEqualTo(foo);
}   // same as bar = foo;
```

The Assignment Operator

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
    m_x = x; m_y = y; m_rad = r;
  }
  Circ &operator= (const Circ &src)
  {
    m_x = src.m_x;
    m_y = src.m_y;
    m_rad = src.m_rad;
    return *this;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

- Ok, here's a nearly complete assignment operator with the proper syntax. It's still missing one thing (an alias check). We'll see that soon.
- If you define this method, it overrides the = operator so when you set:

  circlea = circleb;

  it will call this function instead of just shallow-copying the data byte by byte.
- You must always make the parameter to the assignment operator a const reference (const Circ &).
- The return type of the assignment operator must always be a reference to the class's type (e.g., Circ &) AND the function must always:

  return * this;

  We'll see why in a few slides.
- The body of the assignment operator copies over the values from the parameter to the target variable.

The Assignment Operator

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
    m_x = x; m_y = y; m_rad = r;
  }
  Circ &operator= (const Circ &src)
  {
    m_x = src.m_x;
    m_y = src.m_y;
    m_rad = src.m_rad;
    return *this;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

```
int main()
{
    Circ  foo(1,2,3);

    Circ  bar(4,5,6);

    bar = foo;
}
```

- In this example, the
  bar = foo;
  line will implicitly result in a function call to bar's operator= function, passing in foo as the src variable.
- It's important to note that for a simple class like this, which has no pointers or dynamically-allocated memory, we really don't need to define our own assignment operator.
- We can just use C++'s built-in copy operation, which copies the member variables byte-for-byte from foo over the existing values in bar.
- But we'll see an example next where we must define an assignment operator to properly assign.

The Assignment Operator

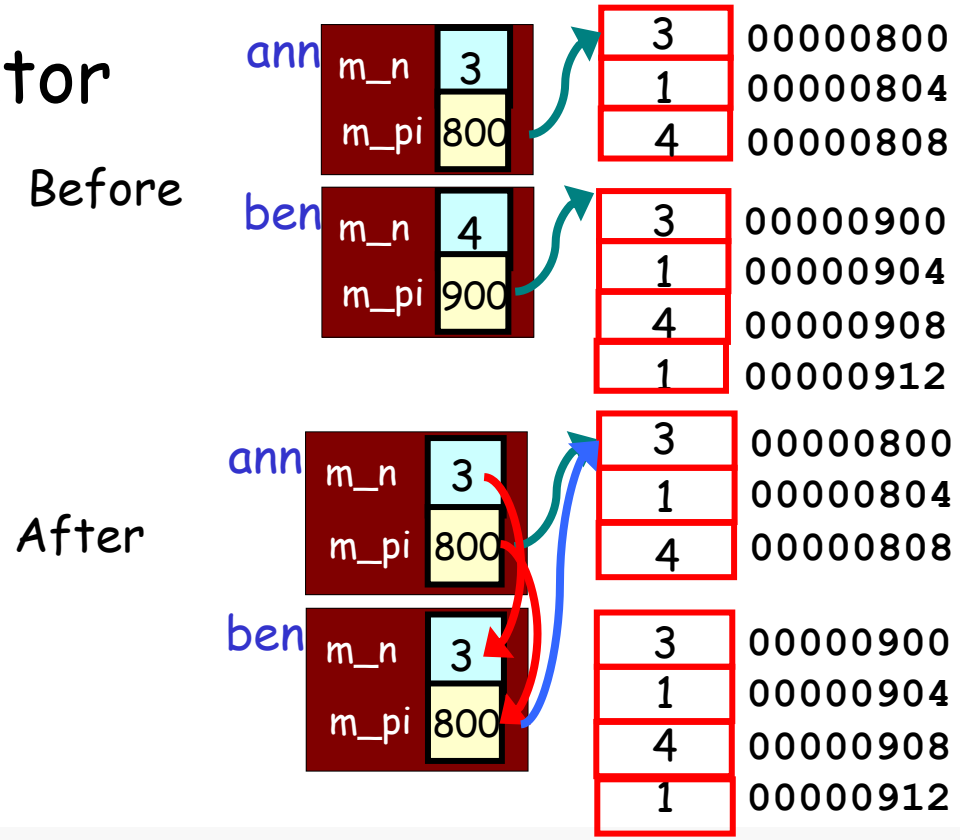
```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];  // #1
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;} // #2

  void showOff()
  {
    for (int j=0;j<n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n; // #3
};
```

- The PiNerd class is an example of a class where we do need to define an assignment operator (and a copy constructor) if we want to do assignments.
- Why? Because its member variables hold pointers to dynamically allocated data (allocated/freed with new a d delete). See lines #1 and #2.
- In general, any time a class has a member variable/pointer to some shared system resource (like a file or network connection), you'll also need an assignment operator.
- If you notice a class has a pointer member variable like on line #3, there's a pretty good chance you'll need an assignment operator (and a copy constructor).

The Assignment Operator

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
 }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<n;j++)
      cout << m_pi[j] << endl;
  }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
    PiNerd  ann(3);

    PiNerd  ben(4);

    ben = ann;
}
```



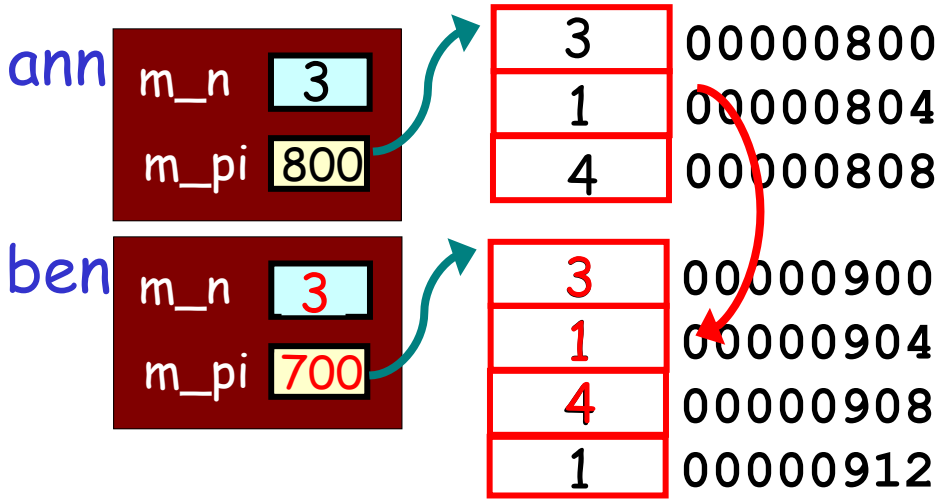
- In this example, we haven't defined an assignment operator. Because PiNerd uses dynamic allocation, that will cause a problem.
- In the **before** diagram, we see the variables ann and ben before we perform a shallow copy with: ben = ann; Everything is fine.
- In the **after** diagram, we see the ann and ben variables after C++ has shallow copied ann's member variables into the ben variable.
- We have two problems after this:
  - Ben and ann now both point to the same array at location 800. When one of the two variables is destructed, it will free the memory that the other is using at 800 as well! That's bad!
  - Ben's old array, which was at 900, is lost – no variable currently points at it anymore, so our ben variable has lost track of it.
  - This will result in a memory leak.

Assignment Operator

For such classes, you must define your own *asignment operator!*

Here's how it works for
ben = ann;

1. Free any memory currently held by the target variable (ben).
2. Determine how much memory is used by the source variable (ann).
3. Allocate the same amount of memory in the target variable.
4. Copy the contents of the source variable to the target variable.
5. Return a reference to the target variable.

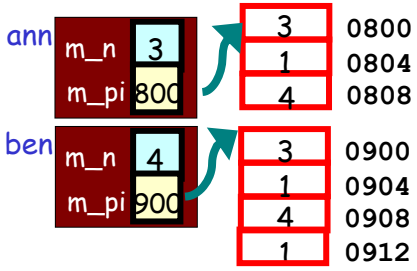


The Assignment Operator

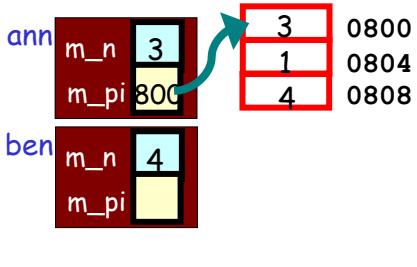
```
class PiNerd
{
public:
  PiNerd(int n) { ... }
  ~PiNerd(){ delete[]m_pi; }

  // assignment operator:
  PiNerd &operator=(const PiNerd &src)
  {
    delete [] m_pi; // #1
    m_n = src.m_n;  // #2
    m_pi = new int[m_n]; // #3
    for (int j=0;j<m_n;j++) // #4
       m_pi[j] = src.m_pi[j];
    return *this; // #5
  }
  void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

- Here's a mostly-complete assignment operator for the PiNerd class (it still doesn't handle aliasing).
- #1: Frees the array of the target PiNerd.
- #2: Copies the size of the array over, so the target array's size will be the same as the src array's size.
- #3: Allocates a new array for the target variable that's the same size as the source array.
- #4: The loop copies the values over from the original array to the target array.
- Finally, the line #5 returns the target object (we'll see why soon)

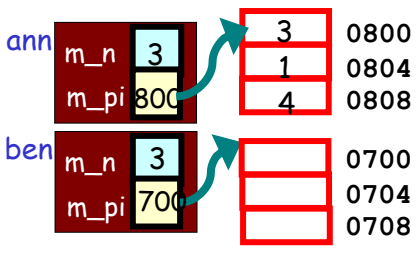
Before ben = ann;



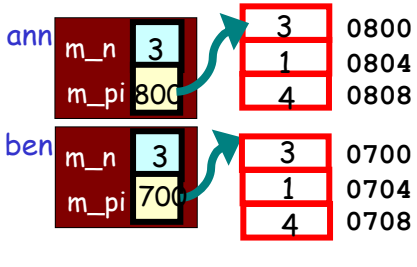
After line #1



After lines #2 and #3



After loop #4

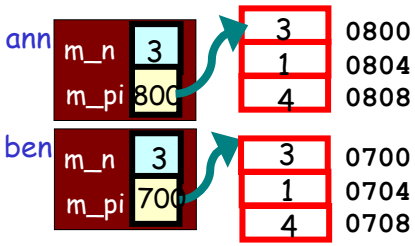


The Assignment Operator

```
class PiNerd
{
public:
  PiNerd(int n) { ... }
  ~PiNerd(){ delete[]m_pi; }

  // assignment operator:
  PiNerd &operator=(const PiNerd &src)
  {
    delete [] m_pi; // #1
    m_n = src.m_n;  // #2
    m_pi = new int[m_n]; // #3
    for (int j=0;j<m_n;j++) // #4
       m_pi[j] = src.m_pi[j];
    return *this;
  }
  void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  PiNerd ben(4);

  ben = ann;

} // ann's d'tor called, then ben's
```

- When we reach the final line of the main function, both our ann and ben variables "go out of scope" and their destructors are called.
- Because both variables point to their own unique array (ann to 800, and ben to 700), when each destructor runs it deletes only the array held by that variable.
- So everything works as planned.



The Assignment Operator

tim

```
class Gassy
{
  Gassy &operator= (const Gassy &src)
  {
    m_age = src.m_age;
    m_ateBeans = src.m_ateBeans;
    return *this;
  }
  m_age 5   m_ateBeans false
};
```

ted

```
class Gassy
{
  Gassy &operator= (const Gassy &src)
  {
    m_age = src.m_age;
    m_ateBeans = src.m_ateBeans;
    return *this;
  }
  m_age 5   m_ateBeans false
};
```

sam

```
class Gassy
{
  Gassy &operator= (const Gassy &src)
  {
    m_age = src.m_age;
    m_ateBeans = src.m_ateBeans;
    return *this;
  }
  m_age 5   m_ateBeans false
};
```

Question: Why do we have return *this at the end of the assignment operator function?

Answer: So we can do multiple assignments in the same statement, like on line #1 below,

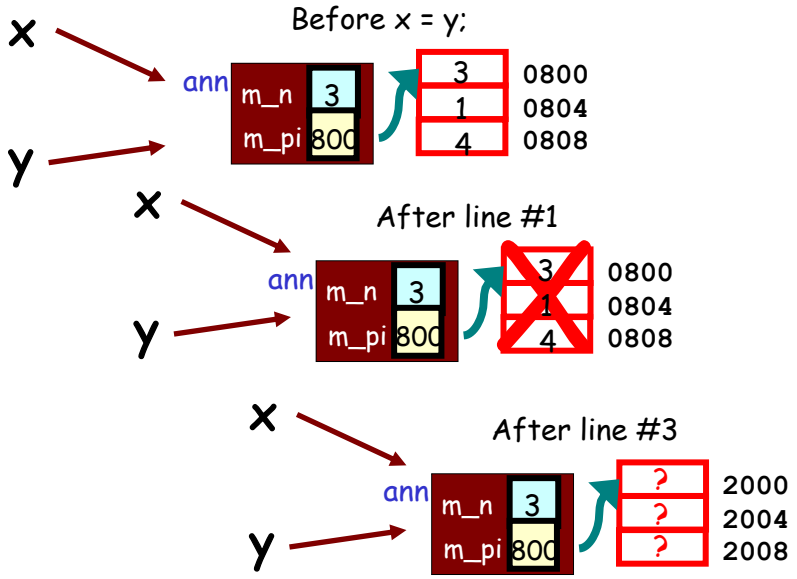
- C++ assigns from right to left, so ted is first assigned to sam, via ted.operator=(sam);
- When the assignment operator for the ted variable finishes, it returns *this. So it's actually returning a reference to the whole ted variable! Crazy, huh? A member function in an object returns the object that holds the member function! Mind blown.
- Then the next assignment occurs, which sets tim equal to what ted.operator=(sam) returned, which is just a reference to ted. So this then runs tim.operator=(ted).
- The result is that ted and tim both contain the same value as sam.

```
int main()
{
  Gassy sam(5,false);
  Gassy ted(10,false);
  Gassy tim(3,true);

  tim = ted = sam; // #1
}
```

The Assignment Operator

```
class PiNerd
{
public:
   PiNerd &operator=(const PiNerd &src)
   {
     delete [] m_pi; // #1
     m_n = src.m_n; // #2
     m_pi = new int[m_n]; // #3
     for (int j=0;j<m_n;j++) // #4
        m_pi[j] = src.m_pi[j];
     return *this;
   }

private:
    int *m_pi, m_n;
};
```

```
void f(PiNerd &x,PiNerd &y)
{
    ...
    x = y; // really ann = ann;  !!!
}

int main()
{
   PiNerd ann(3);
   f(ann,ann);   // #0
}
```



- Our assignment operator has one more problem with it – it doesn't properly handle "aliasing".
- Aliasing is when we use two different pointers/ references to access the same variable.
- Imagine that we pass the ann variable to both parameters of the f() function (line #0)
- x and y both refer to the same ann variable
- When we run line #1, the delete command is supposed to free the memory in the left-hand variable (x). But since x and y both refer to ann, this actually deletes our ONLY copy of the array in ann! We thus deleted y's array too!
- On line #3, we then allocate a new array, which of course starts with random values.
- So after line #4, we copy random values from ann's array (y) to ann's array (x). Badness.

The Assignment Operator

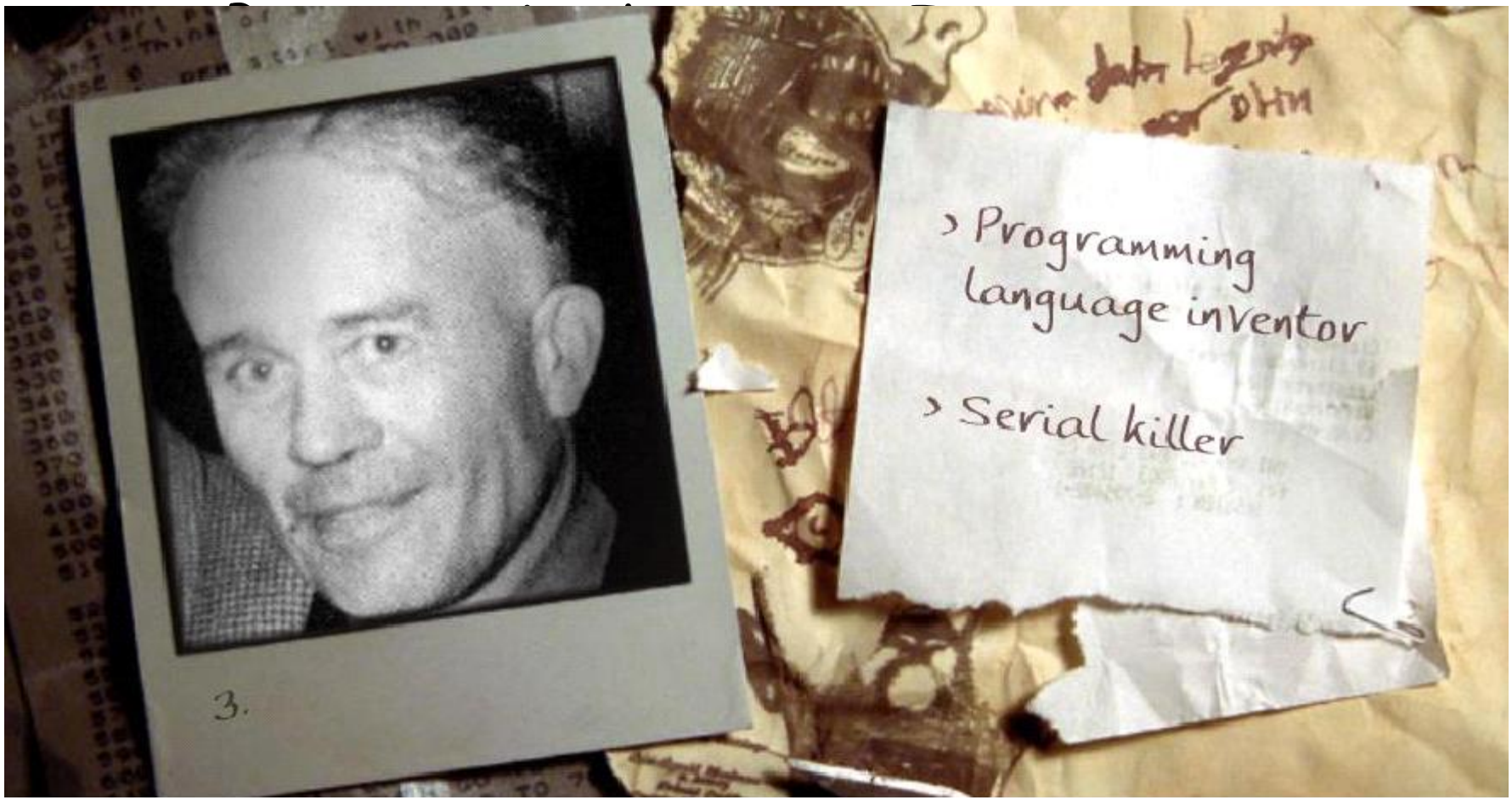
The fix:

Our assignment operator function must check to see if a variable is being assigned to itself, and if so, do nothing…

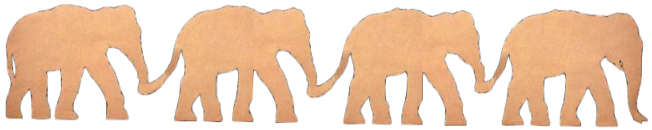
```
class PiNerd
{
  ...
  PiNerd &operator=(const PiNerd &src)
  {
    if (&src == this)  // #1
      return *this; // do nothing
    delete [] m_pi;
    m_n = src.m_n;
    m_pi = new int[m_n];
    for (int j=0;j<m_n;j++)
      m_pi[j] = src.m_pi[j];
    return *this;
  }
  ...
};
```

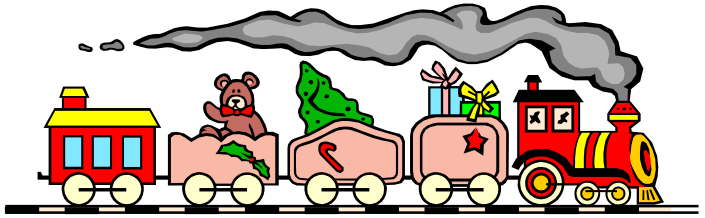
- The solution is to check to see if the parameter (src) has the same address as the target object that's supposed to be changed.
- We can get the target object's address with the this keyword.
- If the target object has the same address as its parameter (line #1) then we're assigning an object to itself (probably via an alias).
- In this case, we don't need to do any assignment.
- Instead, we just return a reference to the target object (or to src – they're the same!) and exit our function.

Time for your favorite game!



Linked Lists

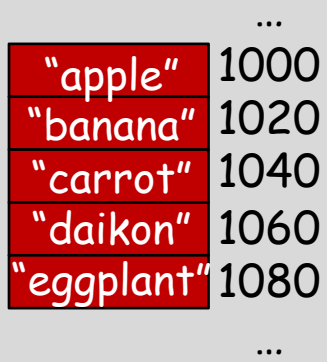




Linked Lists

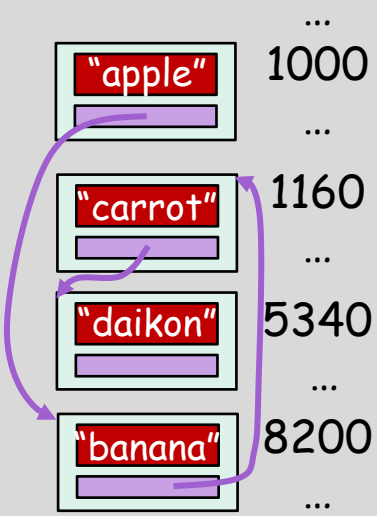
What's the big picture?

An array stores items by reserving a fixed-size, contiguous block of memory up-front.



A linked list reserves a new memory block for each item as it's added, and links blocks together with pointers.

It can hold a variable number of items, which you access by following the pointers.



Arrays are great… But…

Arrays are great when you need to store a fixed number of items…

```
int main()
{
   int array[100];

   …
}
```

But what if you don't know how many items you'll have ahead of time?

```
int main()
{
   // might have 10 items or 1M
   int array[1000000];

    …
}
```

Then you have to reserve enough slots for the largest possible case.

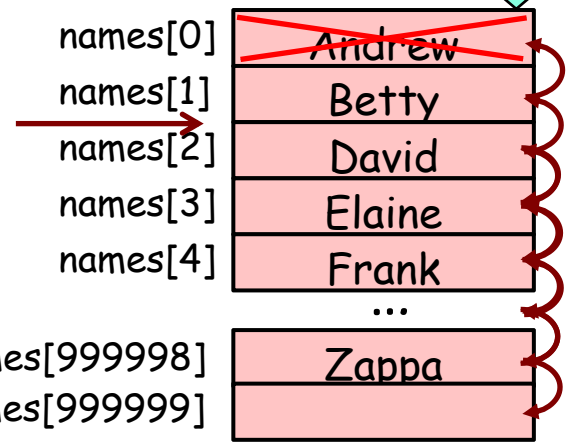
Even new/delete don't really help!

```
int main()
{
  int numItems, *ptr;

  cin >> numItems;
  ptr = new int[numItems];
}
```

It takes nearly 1M steps to add a new item!

And what if you need to insert a new item in the middle of an array?

Carey



We have to move every item below the insertion spot down by one!

And it's just as slow if we want to delete an item! Yuck!

So Arrays Aren't Always Great

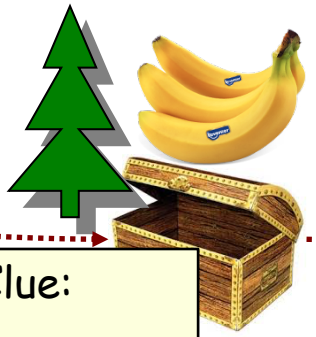
Hmm… Can we think of an approach from "real life" that works better than a fixed-sized array?

How about organizing the items as we would in a Scavenger Hunt?

The hunt starts with a clue to the location of the first chest…

Clue:

The first item is by the tree



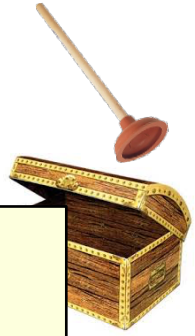
Clue:

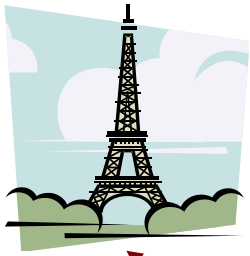
The next item is by the house

Clue:

The next item is by the tower

Then each chest holds an item and a clue to the location of the next chest.





Clue:

This is the last item!

What can we think of that:

allows you to store an arbitrary number of items

makes it fast to insert a new item in the middle

makes it fast to delete an item from the middle

Using this approach we can store an arbitrary number of items!

There's no fixed limit to the number of chests and clues we can have!

A C++ Scavenger Hunt?

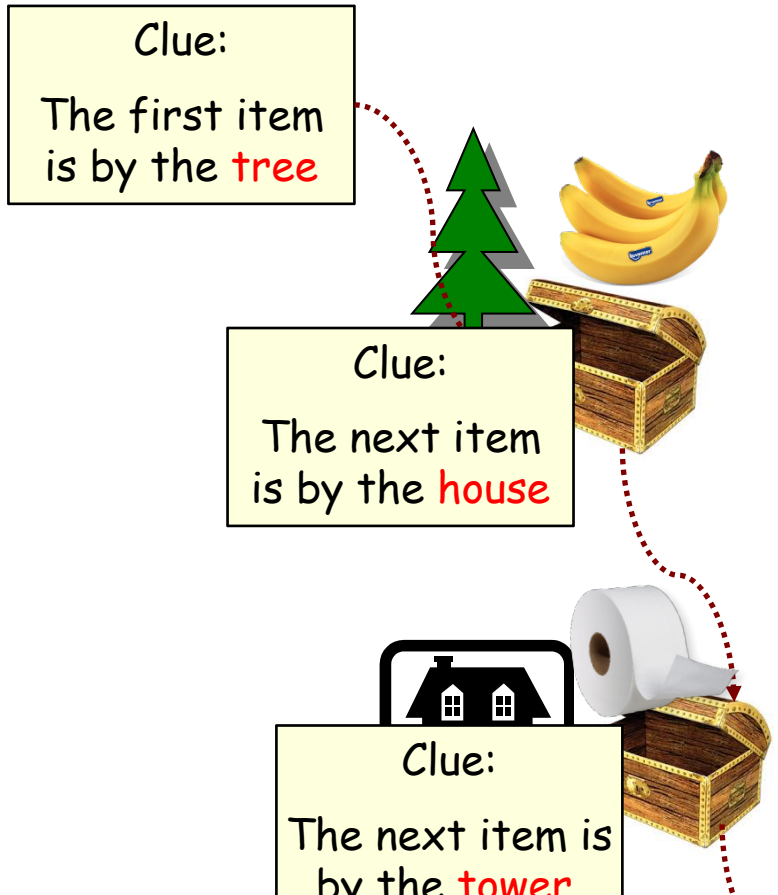
Ok, so in our Scavenger Hunt, we had:

A clue that leads us to our first treasure chest.

Each chest then holds an item (e.g., toast) and a clue that leads us to the next chest.

So here's the question… can we simulate a Scavenger Hunt with a C++ data structure?

Why not? Let's see how.



A C++ Scavenger Hunt?

Well, we can use a C++ struct to represent a Chest.

As we know, each Chest holds two things:

A treasure – let's use a string variable to hold our treasure, e.g., "bacon".

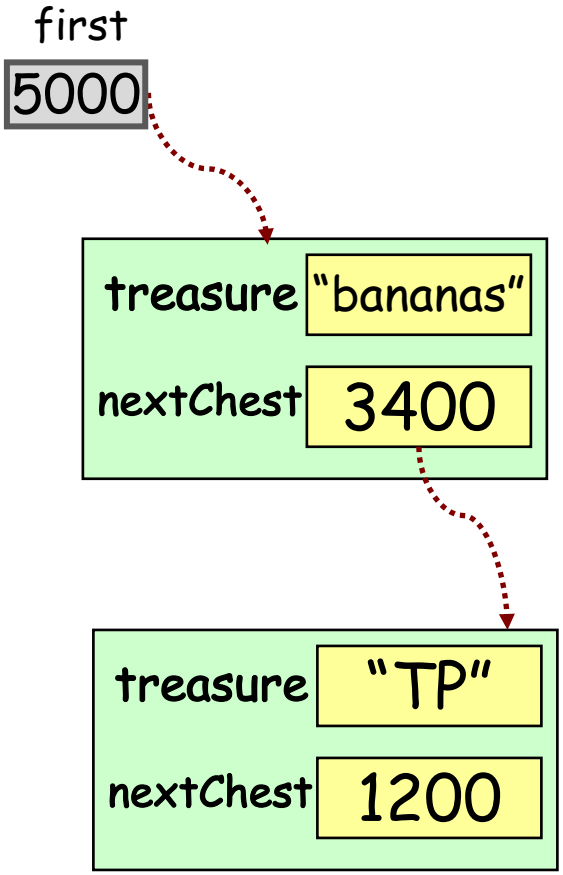
The location of the next chest – let's represent that with a pointer variable.

We can now define a Chest variable for each of the items in our scavenger hunt!

And we can define a pointer to point to the very first chest – our first clue!

```
struct Chest
{
  string treasure;
  Chest * nextChest;
};

// pointer to our 1st chest
Chest *first;
```

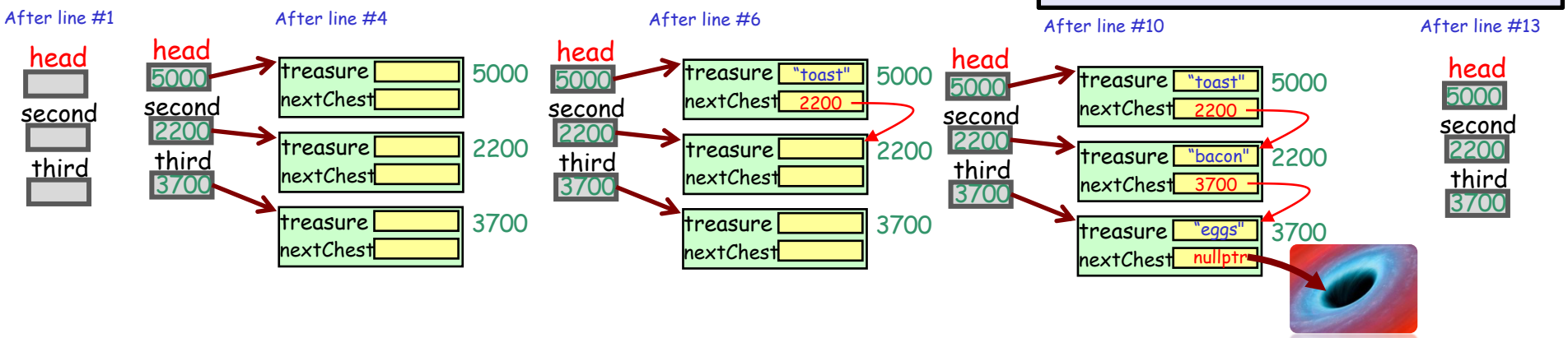
This line basically says that each Chest variable holds a pointer… to another chest variable.



Linked Lists

- On line #1, we define three pointers, which will point to three "chests" in our linked list. The pointer to the top item in the linked list is traditionally called the "head pointer."
- On lines #2 through #4, our code allocates three chest structures using the new command. By line #4, our pointers point to our new empty chests.
- On line #5, we set the treasure in the head Chest to be "toast"
- On line #6, we set the head's nextChest pointer so it points at the second chest (which is at address 2200 in memory). Our first chest is now "linked" to our second chest.
- On lines #7 and #8, we fill in our second chest with bacon and link it to our third chest.
- On lines #9 and #10 we fill in our third chest. Note that on line #10, we set the nextChest pointer to nullptr. This explicitly causes our third chest to be our last chest.
- On lines #11 through #13, we delete our chest structs. This does not delete the pointers themselves, which still hold the addresses of the chest structs. But it does free up the memory occupied by those chest structs.
- Once we've constructed our linked list (but before we've deleted it), starting with just the head pointer, you can reach every element in the list without using your any other external pointers (second and third)!
- We can follow the links in each node to the other nodes.

```
struct Chest
{
   string treasure;
   Chest * nextChest;
};

int main()
{
   Chest *head, *second, *third; // #1

   head = new Chest;  // #2
   second = new Chest;  // #3
   third = new Chest;  // #4

   first->treasure = "toast"; // #5
   first->nextChest = second;  // #6

   second->treasure = "bacon"; // #7
   second->nextChest = third;  // #8

   third->treasure = "eggs"; // #9
   third->nextChest = nullptr;  // #10

   delete head;  // #11
   delete second; // #12
   delete third;  // #13
}
```



Linked Lists

Ok, it's time to start using the right Computer Science terms.

Instead of calling them "chests", let's call each item in the linked list a "Node".

And instead of calling the value held in a node treasure, let's call it "value".

And, instead of calling the linking pointer nextChest, let's call it "next".

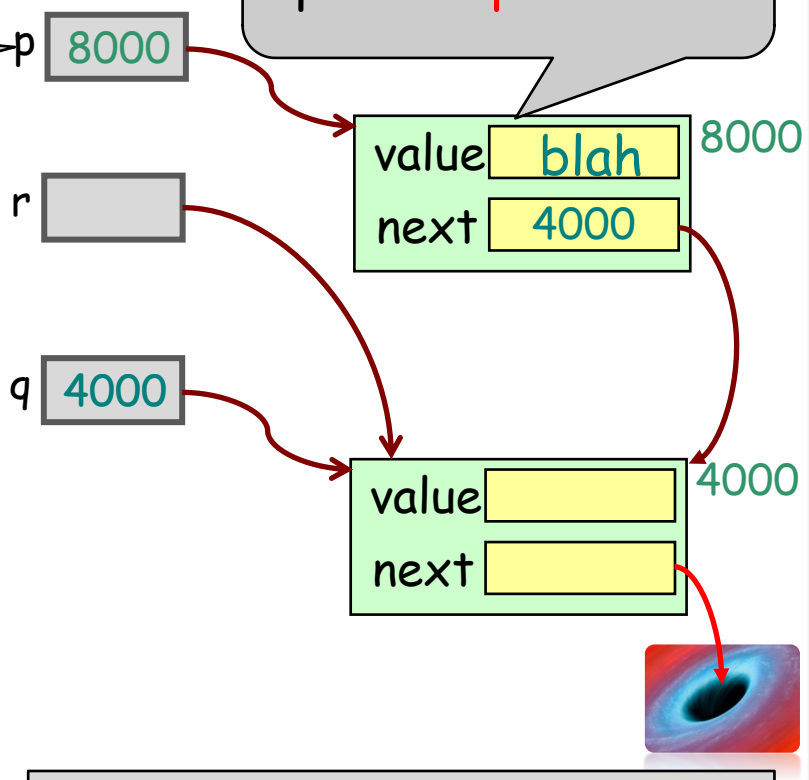
Finally, there's no reason a Node only needs to hold a single value!

```
struct Node // student node
{
    int studentID;
    string name;
    int phoneNumber;
    float gpa;

    Node *next;
};
```

```
struct Node
{
    string value;
    Node * next;
};

int main()
{
  Node *head, *second, *third;

  head = new Node;
  second = new Node;
  third = new Node;

  head->value = "toast";
  head->next = second;

  second->value = "bacon";
  second->next = third;

  third->value = "eggs";
  third->next = nullptr;

  delete head;
  delete second;
  delete third;
}
```

Note: The delete command doesn't kill the pointer...

it kills what the pointer points to!



To allocate new nodes:

```
Node *p = new Node;
Node *q = new Node;
```

To change/access a node p's value:

```
p->value = "blah";
cout << p->value;
```

To make node p link to another node that's at address q:

```
p->next = q;
```

To get the address of the node after p:

```
Node *r = p->next;
```

To make node q a "terminal" node:

```
q->next = nullptr;
```

To free your nodes:

```
delete p;
delete q;
```

Before we continue, here's a short recap on what we've learned:

Linked Lists

Normally, we don't create our linked list all at once in a single function.

After all, some linked lists hold millions of items! That wouldn't fit!

Instead, we create a dedicated class (an ADT) to hold our linked list…

And then add a bunch of member functions to add new items (one at a time), process the items, delete items, etc.

OK, so let's see our new class.

```
struct Node
{
    string value;
    Node * next;
};

int main()
{
  Node *head, *second, *third;

  head = new Node;
  second = new Node;
  third = new Node;

  head->value = "toast";
  head->next = second;

  second->value = "bacon";
  second->next = third;

  third->value = "eggs";
  third->next = nullptr;

  delete head;
  delete second;
  delete third;
}
```

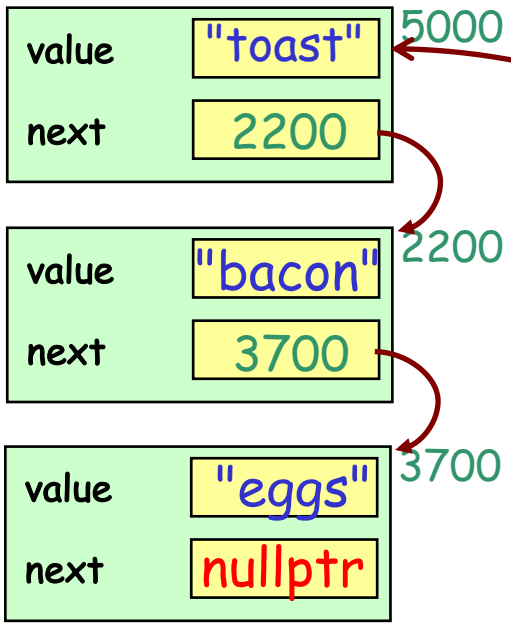
A Linked List Class!

In the simplest type of linked list class, the only member variable we need is a head pointer.

Why? Given just the head pointer, we can follow the links to every node in the list.

And since we can find all the nodes, we can also link in new ones, delete them, etc..

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:

private:
  Node *head;
};
```



Ok, so let's add a head pointer to our class.

5000

A Linked List Class!

Alright, now what methods should our linked list class have?

We need a constructor to create an empty list…

And methods to add new items…

And a method to delete items…

And a method to find if an item is in the list…

And a method to print all the items…

And finally, we need a destructor to free all of our nodes!

Let's consider these one at a time!

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
   LinkedList() { … }
   void addToFront(string v) { … }
   void addToRear(string v) { … }
   void deleteItem(string v) { … }
   bool findItem(string v) { … }
   void printItems() { … }
   ~LinkedList() { … }

private:
  Node *head;
};
```

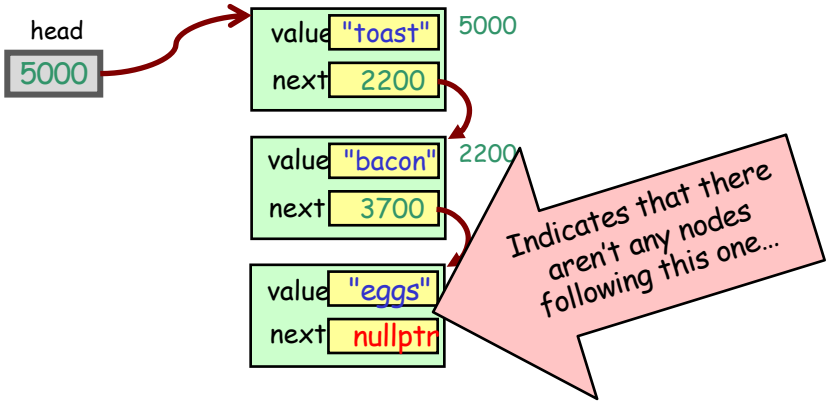
Linked List Constructor

OK, so what should our constructor do?

Well, we'll want it to create an "empty" linked list – one with no items.

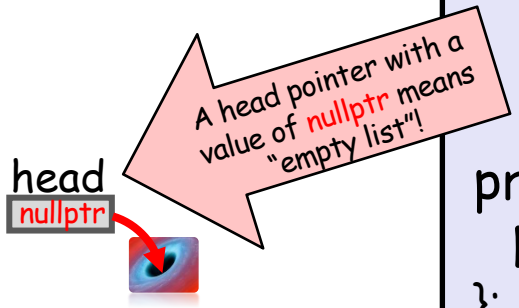
Earlier I showed you how we marked the last node in a linked list…

We set its next value to nullptr.



So, following this logic…

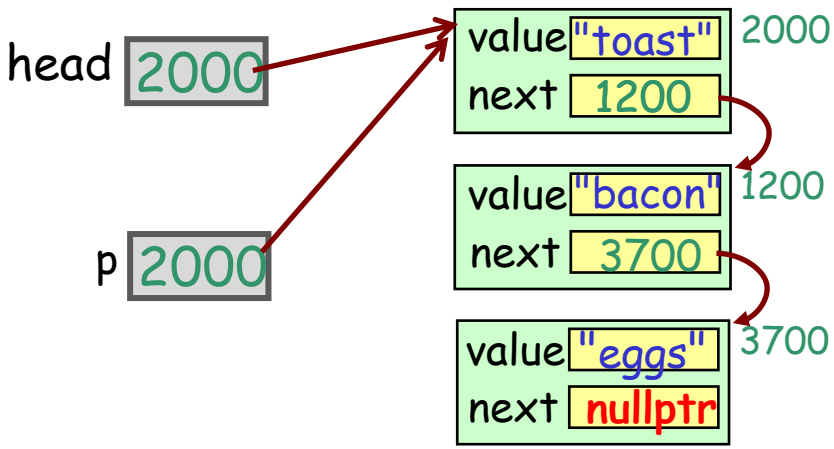
We can create an empty linked list by setting our head pointer to nullptr!




```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
   LinkedList()
   {
      head = nullptr;
   }

   …

private:
   Node *head;
};
```

Printing the Items in a Linked List

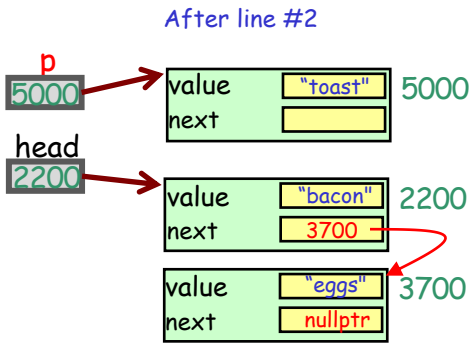
- To print the items in a linked list, we'll use a temporary pointer (p) to iterate through the list.
- Line #1 defines the temp pointer p.
- Line #2 sets p's value equal to the value of the head pointer, so both p and head pointer point at the first node (the node with toast).
- Line #3 loops until p is equal to nullptr. P will be nullptr if either:
  - The list started out empty, and the head pointer is nullptr
  - We've processed every node in the list and just advanced p PAST the last node in the list.
  - In either case, when p is nullptr it means we've printed out all of the items in the linked list, so it's time to stop.
- Line #4 prints out the value in the current node pointed to by p.
- Line #5 advance p to the next node, so if p points at the toast node, p = p->next sets p to 1200, so p points at the bacon node.
- Any time we iterate through one or more nodes like this, it's called a "traversal".

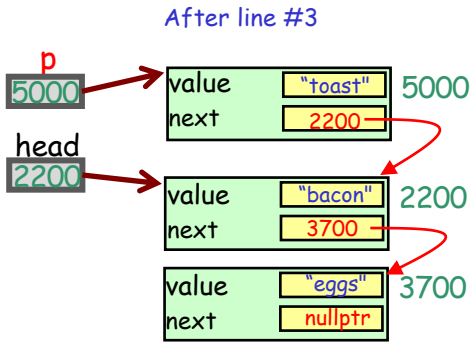


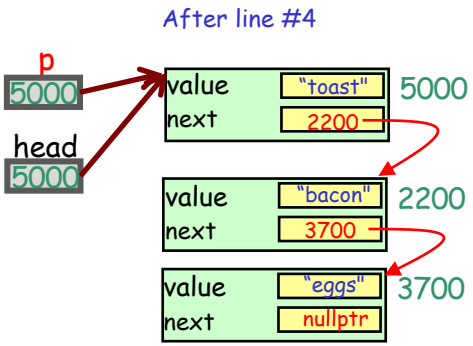

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
    void printItems()
    {
      Node *p;  // #1
      p = head; // #2
      while ( p != nullptr ) // #3
      {
        cout << p->value << endl; // #4
        p = p->next;  // #5
      }
    }


private:
  Node *head;
};
```

Adding an Item to the Front

- Lets add a new item (toast) to the front of our linked list.
- To add a new node to the front of our linked list takes 4 steps.
- The first step (line #1) allocates an empty node to hold our new item that we want to add to the front of the linked list.
- The second step (line #2) puts our item (toast) into the new node.
- The third step (line #3) links our new node to the current head node of the linked list.
- The fourth step (line #4) makes our head pointer point to the newly created node, so it's the first one in the list.
- The new head node points to the old head node in the list.





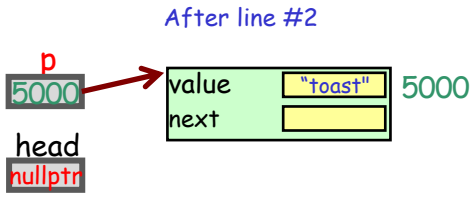


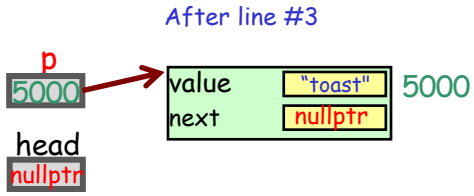

```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
   void addToFront(string v)
   {
      Node *p;
      p = new Node; // #1
      p->value = v;  // #2

      p->next =  head; // #3

      head = p; // #4

   }

   ...



private:
  Node *head;
};
```

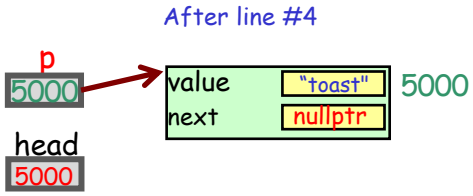
Adding an Item to the Front

OK, but will this same algorithm work if the Linked List is empty?

- Let's find out. Lets add a new item (toast) to the front of an empty linked list (whose head pointer value is nullptr).
- The first step (line #1) allocates an empty node to hold our new item that we want to add to the front of the linked list.
- The second step (line #2) puts our item (toast) into the new node.
- The third step (line #3) links our new node to the current head node of the linked list, which right now is nullptr. So this set's our new node's next pointer to nullptr.
- The fourth step (line #4) makes our head pointer point to the newly created node, so it's the first (and only) one in the list.
- So the same code works whether the link list is empty or already has one or more nodes.








```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
    void addToFront(string v)
    {
      Node *p;
      p = new Node; // #1
      p->value = v;  // #2

      p->next =  head; // #3

      head = p; // #4

    }

    ...


private:
  Node *head;
};
```

Adding an Item to the Rear

Alright, next let's look at how to append an item at the end of a list…

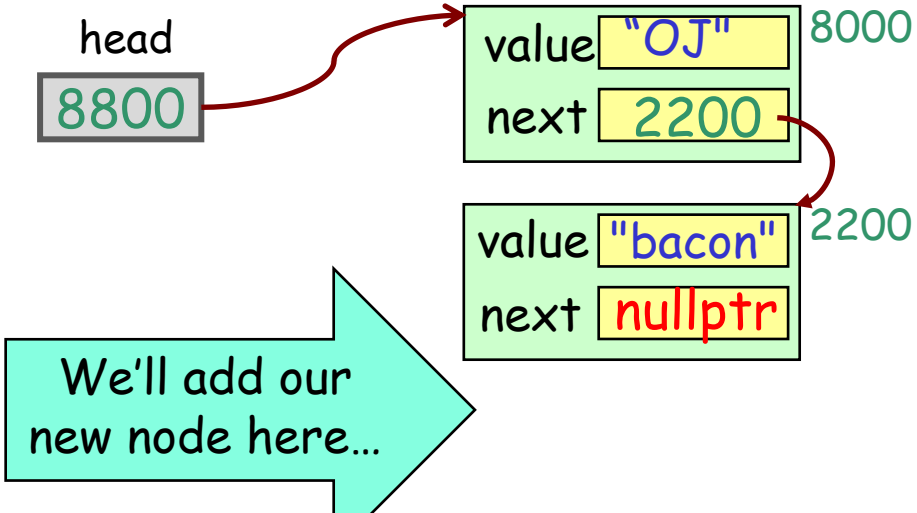
There are actually two cases to consider:

Case #1:

The existing list is totally empty!

Case #2:

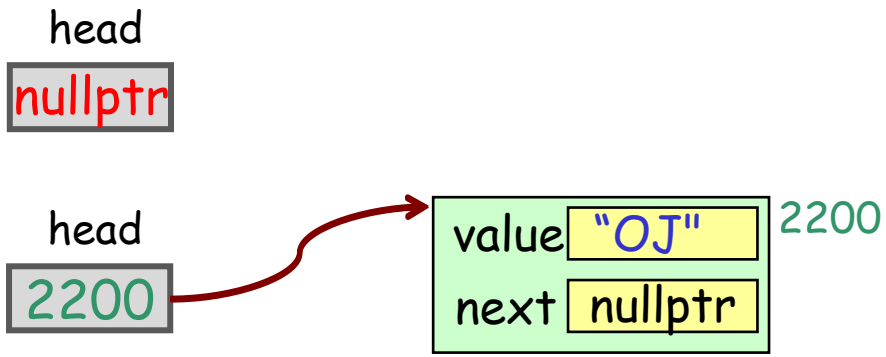
The existing list has one or more nodes…




```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
    void addToRear(string v)
    {
    }
    …
};
```

Adding an Item to the Rear

Alright, let's consider Case #1 first…
It's much easier!



So how do you add a new node to the end of an empty linked list?

In fact, it's the same as adding a new node to the front of an empty linked list.

Which we just learned two minutes ago!

After all, in both cases we're adding a node right at the top of the linked list.

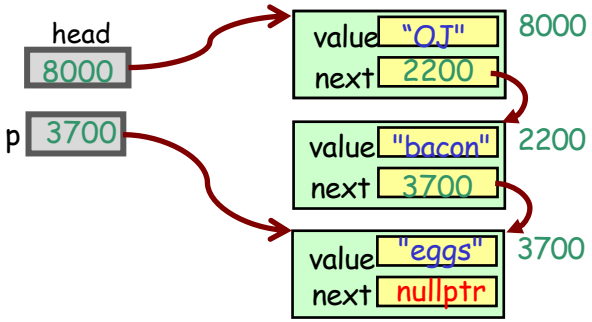
```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
    void addToRear(string v)
    {
       if (head == nullptr)
           addToFront(v); // easy!!!
    }
    …
};
```

Adding an Item to the Rear

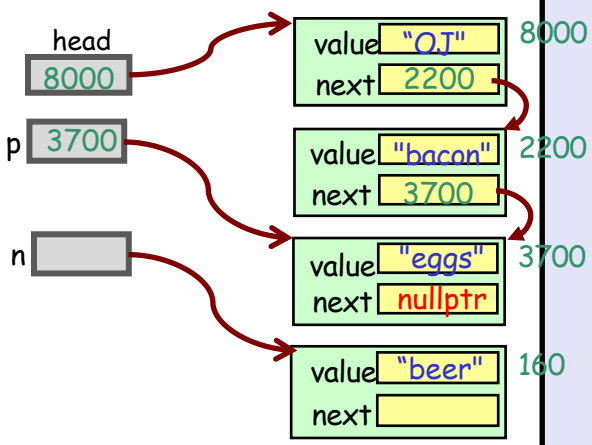
Case #2: Here's the pseudo-code to add an item to the end of a list that has one or more nodes.

Let's add "beer" to the end of our linked list.

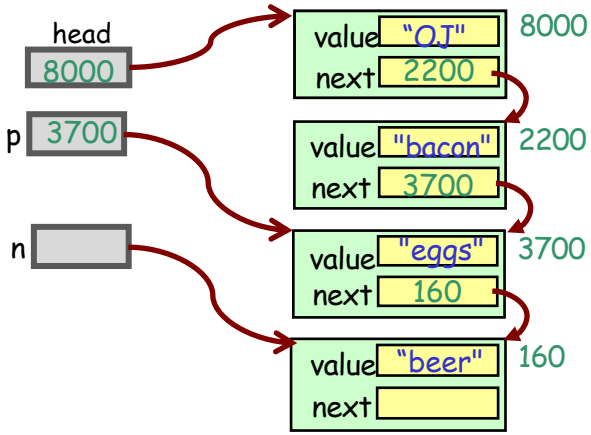
Step #1 uses a loop and a temp variable p to find the last node in the linked list.



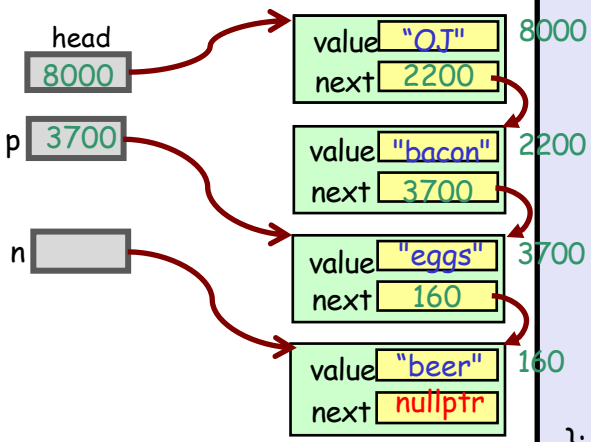
In steps #2 and #3, we allocate a new node and add "beer".



In step #4, we link the current last node to our new node.



In step #5, we set the new node's next pointer to nullptr.

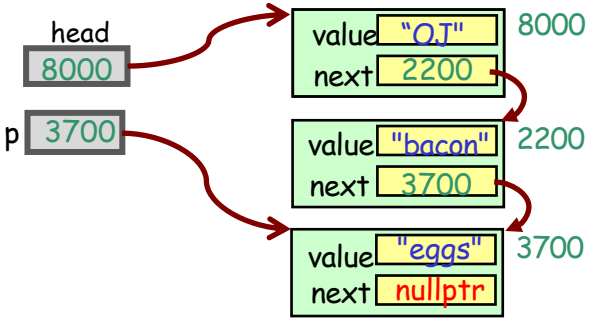



```
struct Node
{
    string value;
    Node   *next;
};

class LinkedList
{
public:

    void addToRear(string v)
    {
        if (head == nullptr)
            addToFront(v); // easy!!!
        else
        {
            Use a temp variable to
            traverse to the current last
            node of the list  (#1)

            Allocate a new node (#2)

            Put value v in the node (#3)

            Link the current last node to our
            new node (#4)

            Link the last node to nullptr (#5)
        }
    }
    ...

};
```

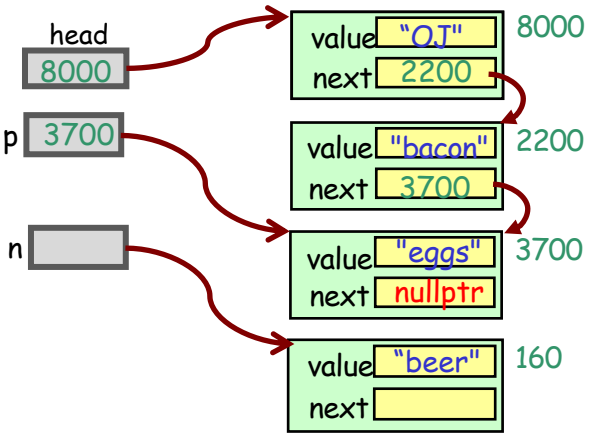
Adding an Item to the Rear

Alright, here's the C++ code for Case #2…

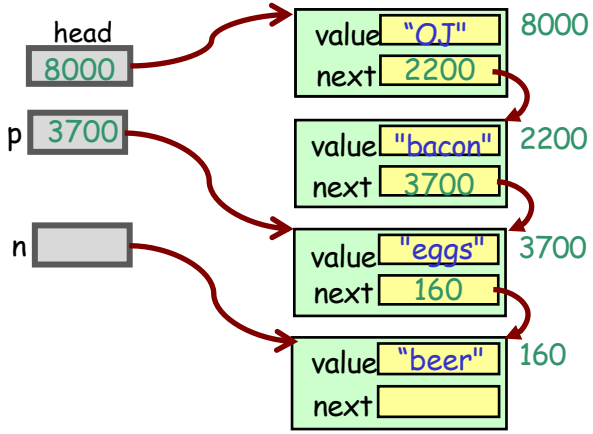
Loop #1 uses p to find to the last node of the list; when the loop ends, p points at the last node in the list.



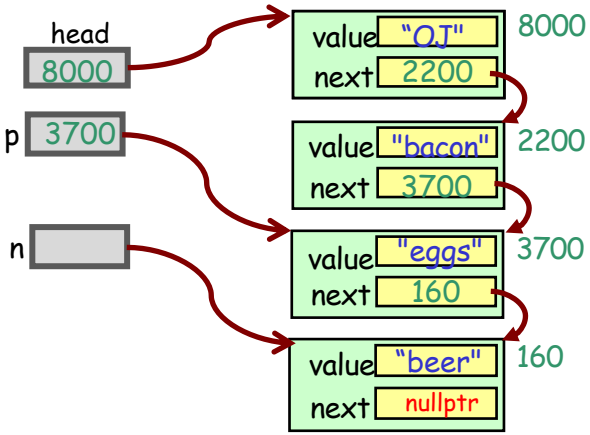
In steps #2 and #3, we allocate a new node and add "beer".



In step #4, we link the current last node to our new node.

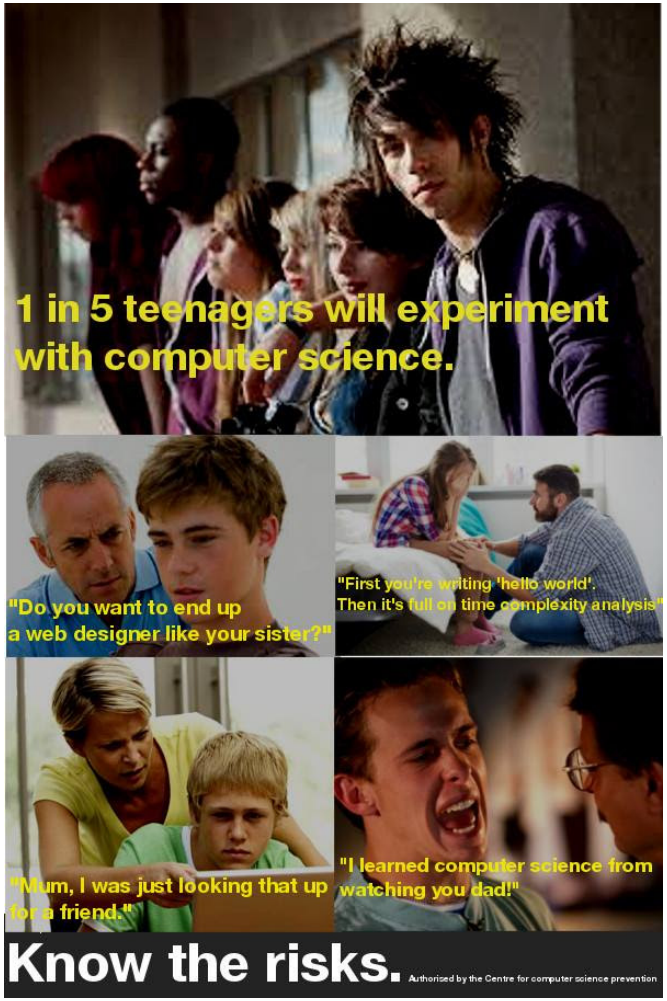


In step #5, we set the new node's next pointer to nullptr.



- Note that this loop , **while (p->next != nullptr)**, is different than the one we used to print things out: **while (p != nullptr)**
  Checking for **p->next != nullptr** ensures that the loop stops when p points AT the last node
- Why? Because p->next is nullptr ONLY if p points at the last node!
- In contrast, **while (p != nullptr)**, only stops looping once p advances PAST the last node and p == nullptr.

```
struct Node
{
    string value;
    Node   *next;
};


class LinkedList
{
public:

    void addToRear(string v)
    {
        if (head == nullptr)

            addToFront(v); // easy!!!

        else
        {
            Node *p;

            p = head;                    // #1
            while(p->next != nullptr) // #1
               p = p->next;              // #1

            Node *n = new Node;  // #2
            n->value = v;  // #3

            p->next =  n; // #4

            n->next = nullptr; // #5
        }

    }
    …

};
```


1 in 5 teenagers will experiment with computer science.
"Do you want to end up a web designer like your sister?"
"First you're writing 'hello world'. Then it's full on time complexity analysis"
"Mum, I was just looking that up for a friend."
"I learned computer science from watching you dad!"
Know the risks.
Authorised by the Centre for computer science prevention

Not at the top, not at the bottom…

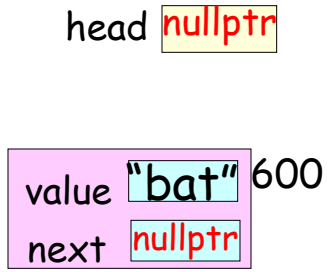
In some cases, we won't always want to just add our node to the top or bottom of the list… Why?

Well, what if we have to maintain an alphabetized linked list, or we want to allow the user to pick the spot to put each item?

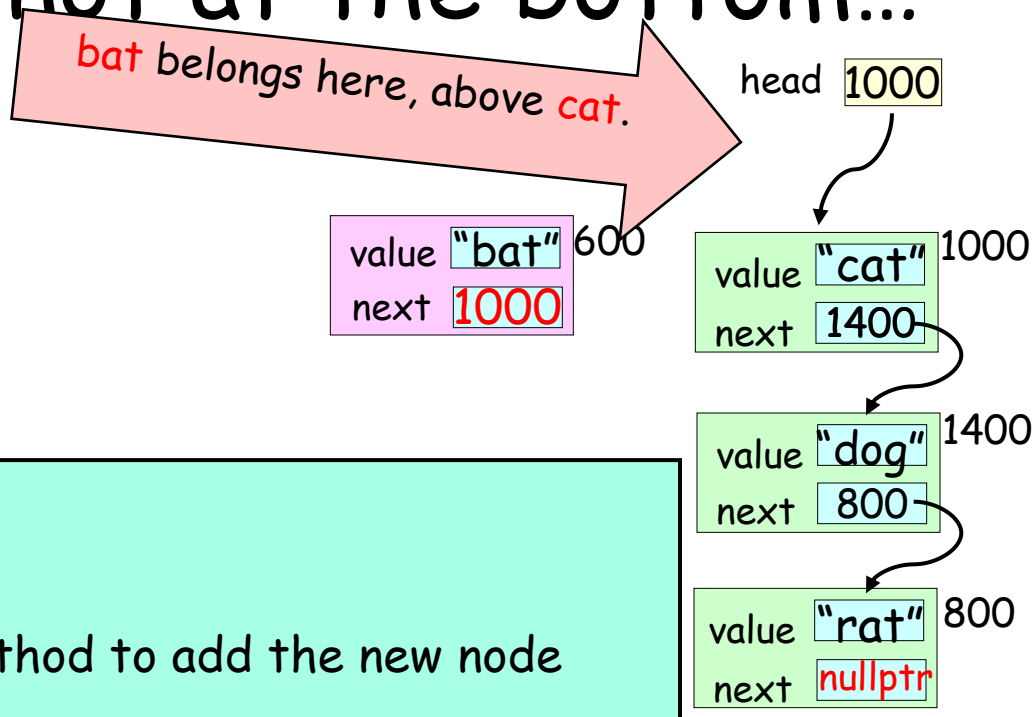
In these cases, we can't just add new items at the top or bottom…

Here's the basic algorithm:

```
void AddItem(string newItem)
{
    if (our list is totally empty)
        Just use our addToFront() method to add the new node









}
```



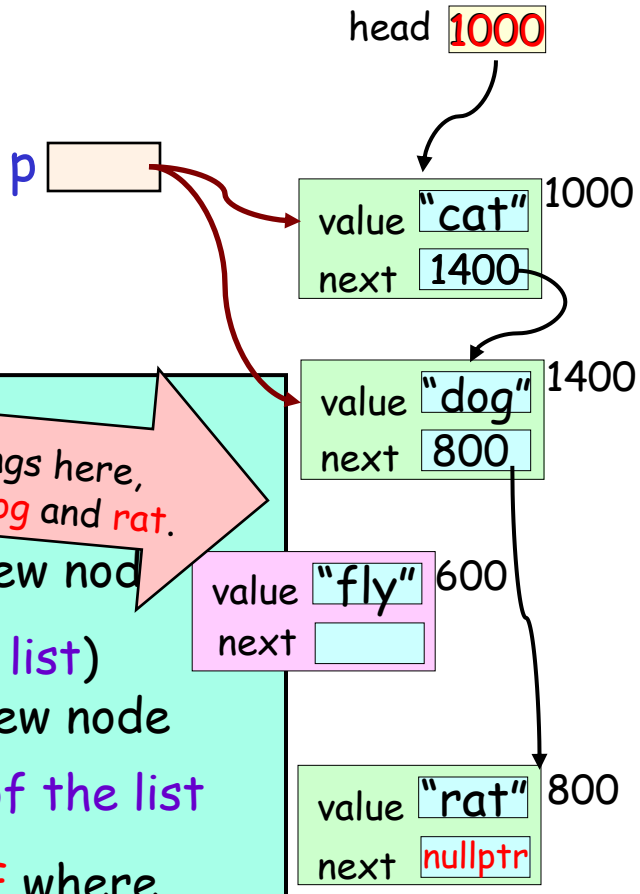
Not at the top, not at the bottom…



Here's the basic algorithm:

```
void AddItem(string newItem)
{
     if (our list is totally empty)
          Just use our addToFront() method to add the new node

     else if (our new node belongs at the very top of the list)
          Just use our addToFront() method to add the new node









}
```

Not at the top, not at the bottom…

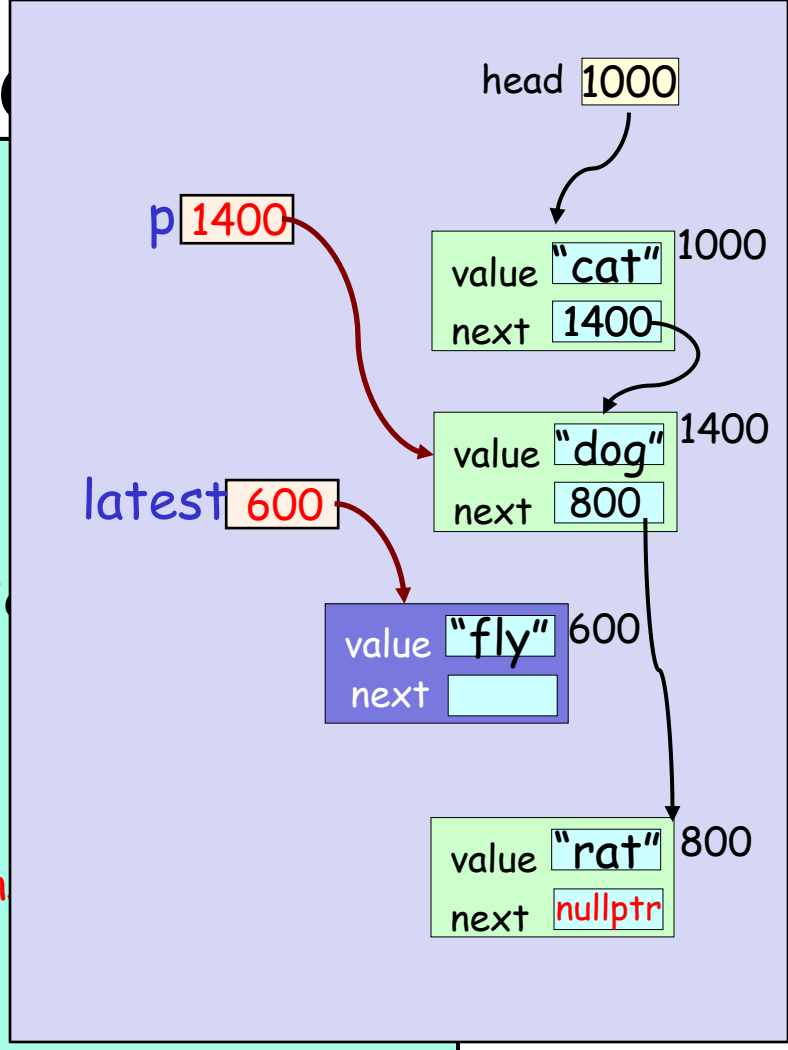


Here's the basic algorithm:

```
void AddItem(string newItem)
{
    if (our list is totally empty)
        Just use our addToFront() method to add the new nod

    else if (our new node belongs at the very top of the list)
        Just use our addToFront() method to add the new node

    else // new node belongs somewhere in the middle of the list
    {
        Use a traversal loop to find the node just ABOVE where
        you want to insert our new item

        Allocate and fill our new node with the item

        Link the new node into the list right after the ABOVE node
    }
}
```

Let's Convert it t

```
void AddItem(string newItem)
{
    if (head == nullptr)
        AddToFront(newItem);

    else if ( /* decide if the new item belongs at the top */ )
        AddToFront(newItem);

    else // new node belongs somewhere in the mi
    {
        Node *p = head;   // start with top node
        while (p->next != nullptr)
        {
            if (/* p points just above where I want to in
                break;   // break out of the loop!

            p = p->next;     // move down one node
        }
        Node *latest = new Node;    // alloc and fill our new node
        latest->value = newItem;
        latest->next = p->next;   // link new node to the node below
        p->next = latest;         // link node above to our new node
    }
}
```



These two lines must be in this order!

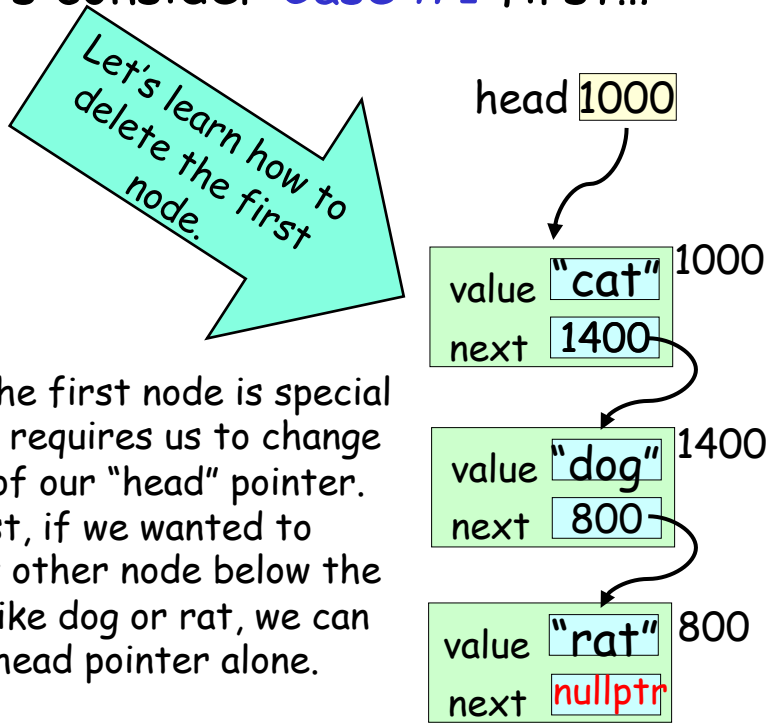
Deleting an Item in a Linked List

When deleting an item from a linked list, there are two different cases to consider:

Case #1: You're deleting the first node.

Case #2: You're deleting an interior node or the last node.

Let's consider Case #1 first...



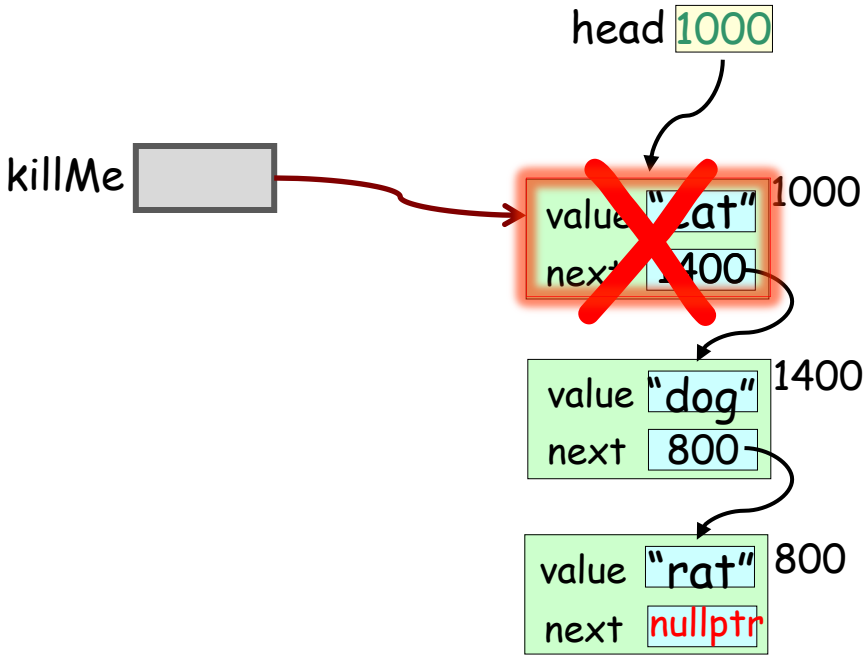
- Deleting the first node is special because it requires us to change the value of our "head" pointer.
- In contrast, if we wanted to delete any other node below the first one like dog or rat, we can leave the head pointer alone.

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
    void deleteItem(string v)
    {




    }
    ...
};
```

Deleting an Item in a Linked List

Ok, let's consider Case #1…
deleting the top item in a list.

Let's kill our cat.

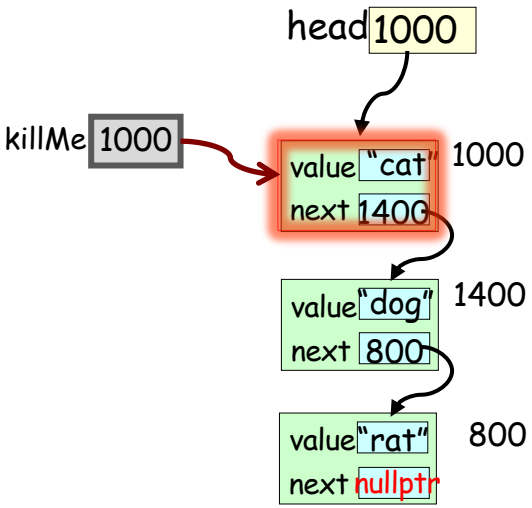



```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
    void deleteItem(string v)
    {
      If the list's empty then return
      If the first node holds the
       item we wish to delete then
      {
        killMe = address of top node
        Update head to point to the
        second node in the list
        Delete our target node
        Return – we're done
      }


    }
    …
};
```

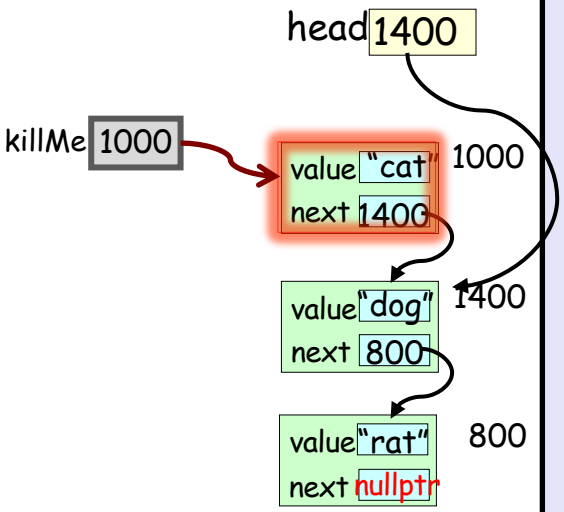
Deleting an Item in a Linked List

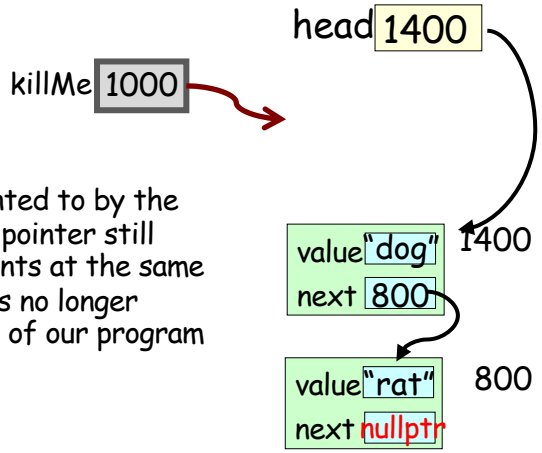
- On line #1 we check to see if the item we want to delete (cat) is held in the first node, pointed to by head.
- If so, we proceed with deleting the first node. See below.

On line #2: killMe points at our top node that we want to delete.



On line #3: We change our head pointer to point to the node after the killMe node.



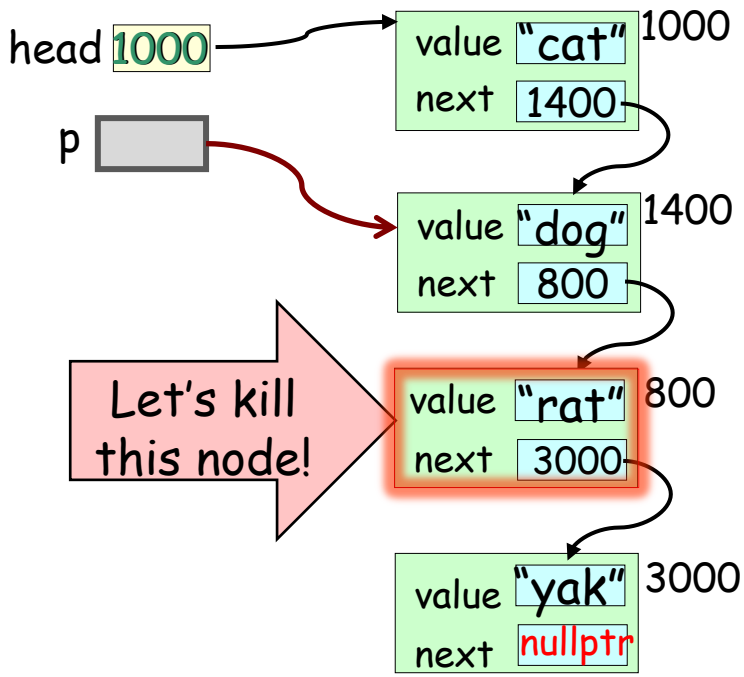


On line #4: We delete the node pointed to by the killMe pointer. Note that our killMe pointer still contains a value of 1000 and still points at the same place. But the node that was there is no longer owned by our code. Some other part of our program may own this memory space now.

```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:                                   "cat"
    void deleteItem(string v)
    {
      if (head == nullptr) return;

      if (head->value == v) // #1

        Node *killMe = head; // #2

        head = killMe->next;  // #3

        delete killMe;  // #4

        return;



    }
    ...
};
```

Deleting an Item in a Linked List

- Now let's see how to delete an item from the middle or end of the list, e.g., "rat"
- To the right is the pseudo-code for this.

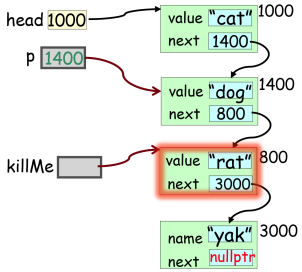



```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
                                        "rat"
    void deleteItem(string v)
    {
      … // the code we just wrote
      Use a temp pointer to traverse
      down to the node above the
      one we want to delete…




      If we found our target node
      {
        killMe = addr of target node
        Link the node above to
        the node below
        Delete our target node
      }
    }
    …
};
```

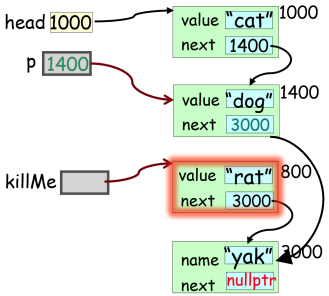
Deleting an Item in a Linked List

- Here's the C++ code for deleting a middle/end node.
- The while loop on line #1 traverses down the linked list until it finds the node we want to delete (e.g. "rat")
- Notice that the loop starts p pointing at the head node, but always checks the node below the one p points at:
  if (p->next != nullptr && p->next->value == v)
- If we find the target value in p->next->value, then p will point to the node just above it (at "dog" in this example)
- When we hit line #3, we begin the deletion if p points at a valid node (or abort if p is nullptr because the value wasn't found).

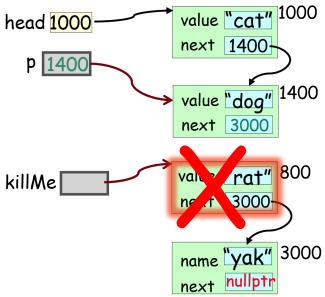
On line #4 we create a new pointer, killMe, to point at our target node.



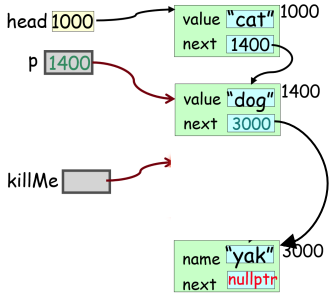
On line #5 we change the node above our target that p points at (dog) so its next pointer points to the node below our target (at yak).



On line #6 we delete our target node.



Our final result; "rat" was deleted.




```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
                                        "rat"
    void deleteItem(string v)
    {
      … // the code we just wrote
      Node *p = head;
      while (p != nullptr) // #1
      {
         if ( p->next != nullptr   &&
              p->next->value == v ) // #2
            break;  // p pts to node above
         p = p->next;
      }
      if (p != nullptr) // #3
         Node *killMe = p->next; // #4
         p->next =  killMe->next; // #5
         delete killMe; // #6
    }
    …
};
```

Linked List Challenge

Now it's your turn!

How would you write the findItem() method?

It should return true if it can find the passed-in item, and false otherwise.

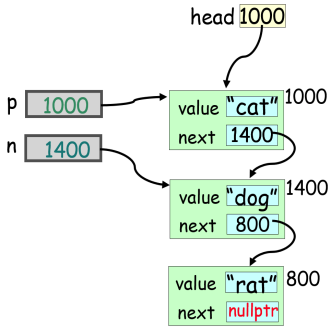
```
int main()
{
    Linked List myFriends;

    myFriends.addToFront("David");
    …

    if (myFriends.findItem("Carey") == true)
        cout << "I'm so lucky!\n";
}
```

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
    bool findItem(string v)
    {









    }
    …


private:
  Node *head;
};
```

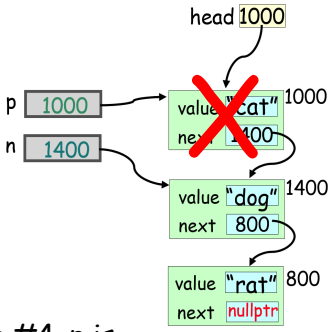
Destructing a Linked List

- Here's the code to delete all of the nodes in a linked list.
- As you can see, it uses a simple traversal which starts by pointing pointer p at the head node (line #1)
- We keep looping until p == nullptr, meaning we loop until we go through each node AND past the last node.
- During each iteration of the loop:
  - Line #1 gets a pointer to the node after the current one pointed at by p, so we don't forget where it is.
  - Line #2 deletes the node pointed at by p.
  - Line #3 points p at the node below the one that was just deleted, so we can continue deleting.

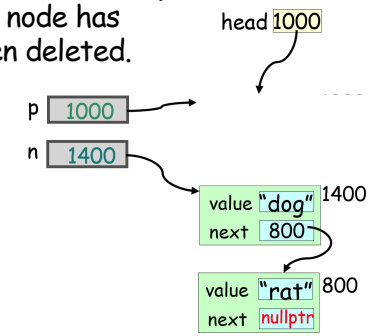
On line #2 we set n so it points at the node below the current node that we're about to delete.



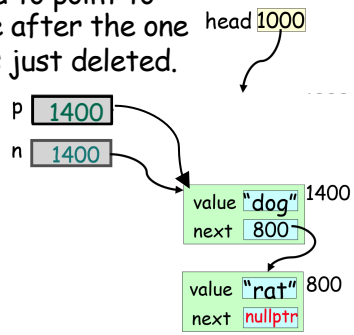
On line #3 we delete the current node pointed at by p.



After line #3 (before line #4) our node has been deleted.



On line #4, p is advanced to point to the node after the one that was just deleted.




```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
   ~LinkedList()
   {
      Node *p;
      p = head; // #1
      while (p != nullptr)
      {
         Node *n = p->next; // #2
         delete p; // #3
         p = n; // #4
      }
   }
   ...


private:
  Node *head;
};
```

Linked Lists Aren't Perfect!

As you can already tell, linked lists aren't perfect either!

First of all, they're much more complex than arrays!!!

Second, to access the k^{th} item, I have to traverse down k-1 times from the head first! No instant access!!!

And to add an item at the end of the list… I have to traverse through all N existing nodes first!

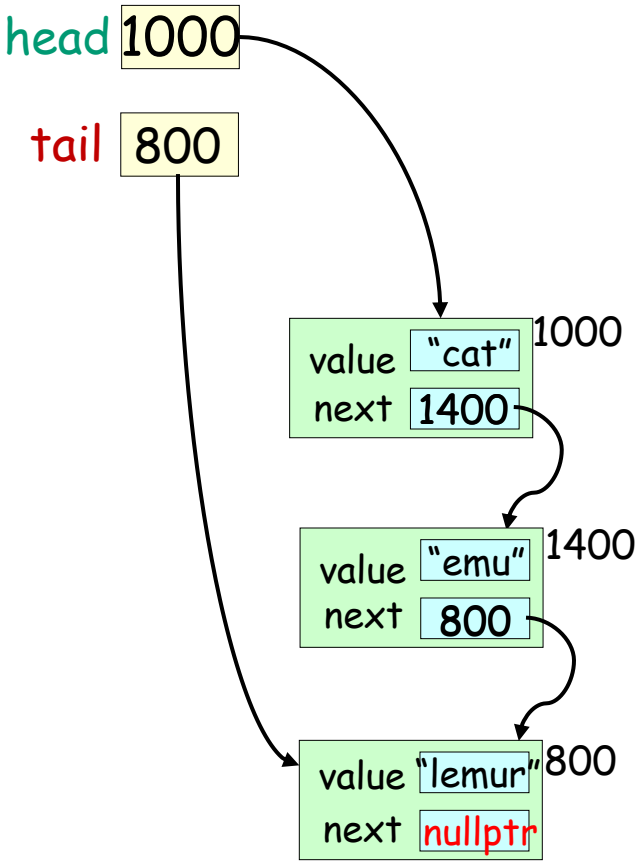
Well, as it turns out, we can fix this last problem… Let's see how!

Linked Lists and Tail Pointers

Since we have a head pointer…

Why not maintain a "tail" pointer too?

A tail pointer is a pointer that always points to the last node of the list!



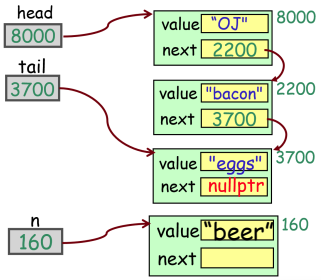

```
class LinkedList
{
public:
   LinkedList() {…}
   void addToFront(string v) {…}
   …

private:
  Node  *head;
  Node  *tail;
};
```

Using the tail pointer, we can add new items to the end of our list without traversing!

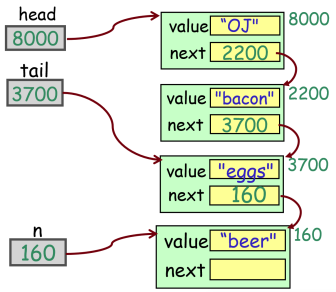
Adding an Item to the Rear… With a Tail Pointer

- Here's the C++ code for adding an item to the rear of a linked list that uses a tail pointer.
- Note that you'll also have to update your delete() method, addToFront() method, etc. addToRear() is just one example.
- As before, if the linked list is empty (head == nullptr), then we call addToFront() to add our new node.
- Otherwise…

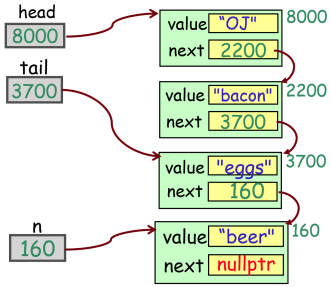
On lines #1 and #2: We allocate a new node (pointed to by n) and set its value.



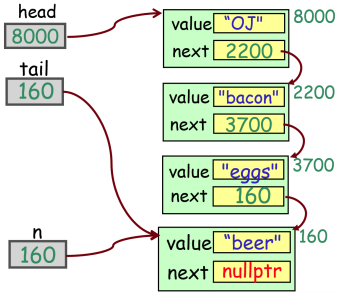
On line #3: We set the current tail node's next pointer (egg's tail->next) equal to the location of our new node (n).



On line #4: We set our new node's next pointer to nullptr, so it becomes the new last node in the linked list.



On line #5: We change our tail pointer so it points at our new last node (rather than point at the old last node "eggs").




```
class LinkedList
{
public:
   void addToRear(string v)
   {
      if (head == nullptr)
         addToFront(v);
      else
      {
         Node *n = new Node; // #1
         n->value = v;   // #2
         tail->next = n; // #3
         n->next = nullptr;  // #4
         tail = n;  // #5
      }
   }
   …

   …
private:
   Node  *head;
   Node  *tail;
};
```

Doubly-linked Lists

One of the downsides with our simple linked list is that we can only travel in one direction… down!

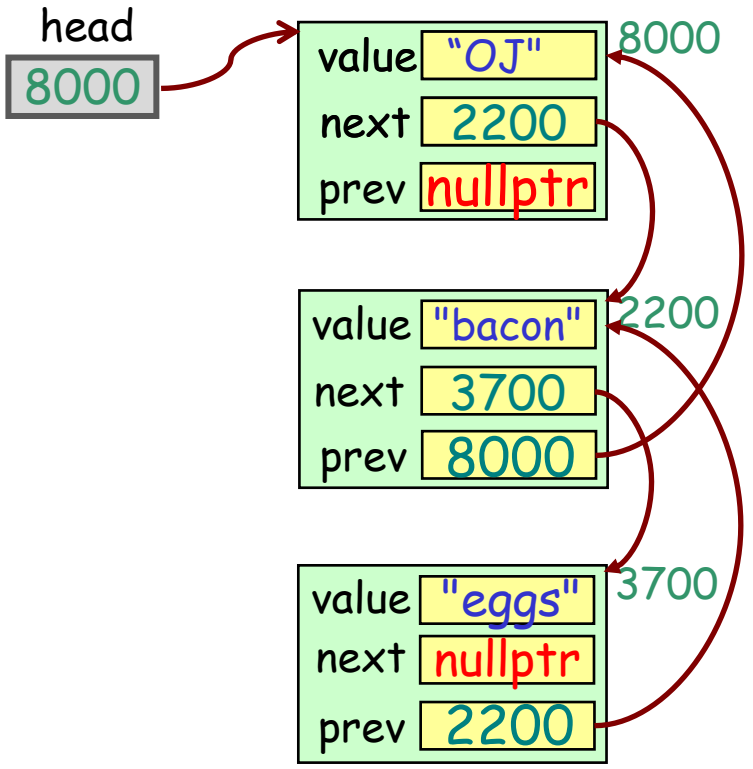
Given a pointer to a node, I can only find nodes below it!

Wouldn't it be nice if we could move both directions in a linked list?

We can! With a doubly-linked list!

A doubly-linked list has both *next* and *previous* pointers in every node:

```
struct Node
{
    string   value;

    Node *   next;
    Node *   prev;
};
```



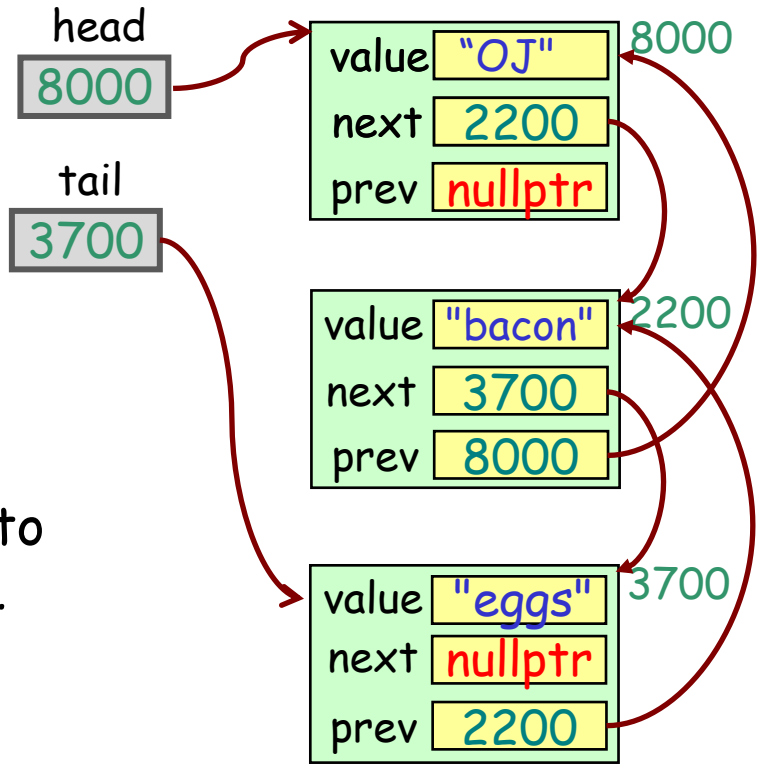
Doubly-linked Lists

And, if I like, I can have a tail pointer too!

Now I can traverse in both directions!

```
Node *p;

p = head;
while (p != nullptr)
{
    cout << p->value;
    p = p->next;
}
```

```
Node *p;

p = tail;
while (p != nullptr)
{
    cout << p->value;
    p = p->prev;
}
```



Of course, now we're going to have to link up lots of additional pointers...

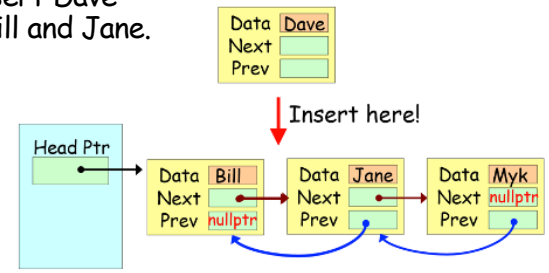
But nothing comes free in life! ☺

Doubly-linked Lists: What Changes?

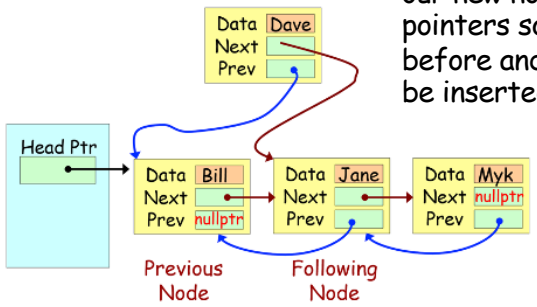
Every time we insert a new node or delete an existing node, we must update *three* sets of pointers:

1. The new node's next and previous pointers.
2. The previous node's next pointer.
3. The following node's previous pointer.

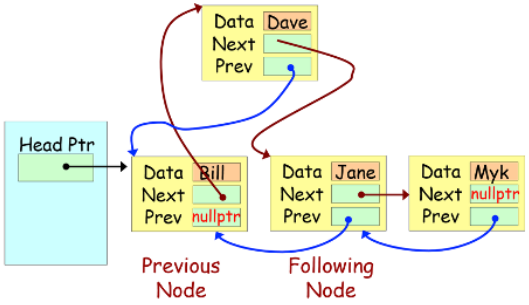
Before Step #0: We want to insert Dave between Bill and Jane.



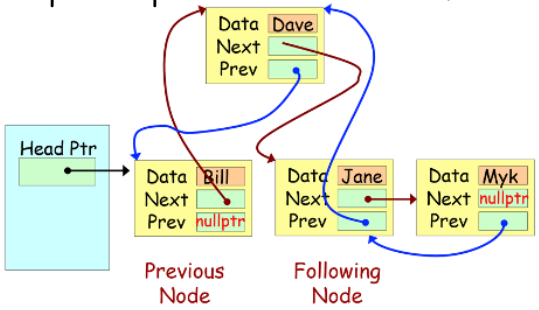
After Step #1: We've updated our new node's next/prev pointers so it points to the node before and after where it is to be inserted.



After Step #2: We've updated the node before (Bill) so its next pointer points to our new node.



After Step #3: We've updated the node after (Jane) so its prev pointer points to our new node.



And of course, we still have special cases if we insert or delete nodes at the top or the bottom of the list.

Linked List Cheat Sheet

```
struct Node
{
    string value;

    Node *next;
    Node  *prev;
};
```

Given a pointer to a node: Node *ptr;

NEVER access a node's data until validating its pointer:

```
if (ptr != nullptr)
    cout << ptr->value;
```

To advance ptr to the next node/end of the list:

```
if (ptr != nullptr)
    ptr = ptr->next;
```

To see if ptr points to the last node in a list:

```
if (ptr != nullptr && ptr->next == nullptr)
    then-ptr-points-to-last-node;
```

To get to the next node's data:

```
if (ptr != nullptr && ptr->next != nullptr)
    cout << ptr->next->value;
```

To get the head node's data:

```
if (head != nullptr)
    cout << head->value;
```

To check if a list is empty:

```
if (head == nullptr)
    cout << "List is empty";
```

Does our traversal meet this requirement?

```
NODE *ptr = head;
while (ptr != nullptr)
{
   cout << ptr->value;
   ptr = ptr->next;
}
```

To check if a pointer points to the first node in a list:

```
if (ptr == head)
    cout << "ptr is first node";
```

Linked Lists vs. Arrays

Which is Faster?

Getting to the 753rd item in a linked list or an array?

Winner: Array

We can get to any item in an array in 1 step. We have to pass thru 752 other nodes to reach the 753rd node in a list!

Which is Faster?

Inserting a new item at the front of a linked list or at the front of an array?

Winner: Linked List

We can insert a new item in a few steps! With an array, we'd have to shift all n items down first!

Which is faster?

Removing an item from the middle of a linked list or the middle of an array?

Winner: Linked List

Once we've found the item we want to delete, we can remove it in a few steps! With an array, we'd have to shift all the following items up one slot!

Which is easier to program?

Which data structure will take less time to program and debug?

Winner: Array

Let's face it – arrays are easier to use. So only use a linked list if you really have to!

Class Challenge

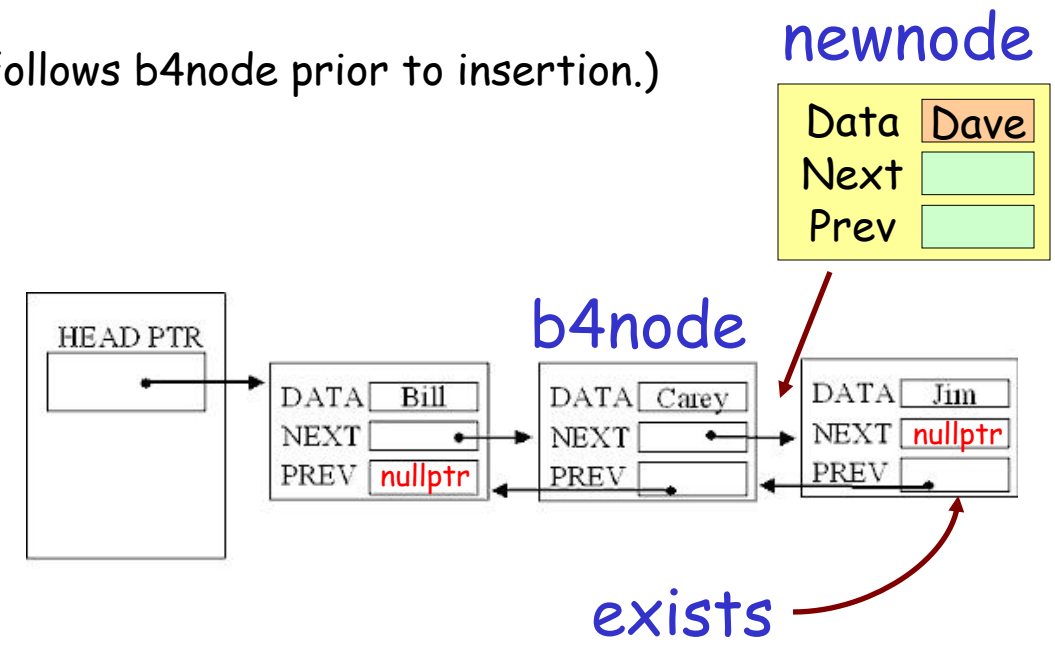
Write a function called insert that accepts two NODE pointers as arguments:

b4node: points to a node in a doubly-linked list
newnode: points to a new node you want to insert

When your function is called, it should insert newnode *after* b4node in the list, properly linking all nodes.

(You may assume that a valid node follows b4node prior to insertion.)

```
struct NODE
{
  string data;
  NODE *next, *prev;
};
```



Appendix: On Your Own Study

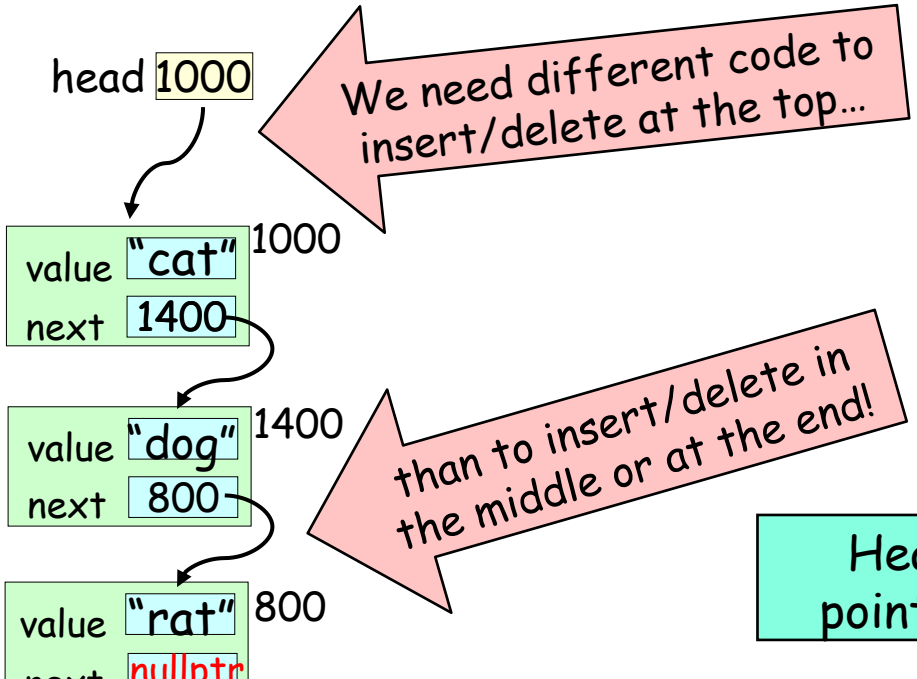
- Linked Lists with Dummy Nodes!

Linked Lists with a Dummy Node

So far, every linked list we've seen has had a head pointer.

But as we've seen this causes complications…

We can simplify things by replacing our header pointer with a dummy node!



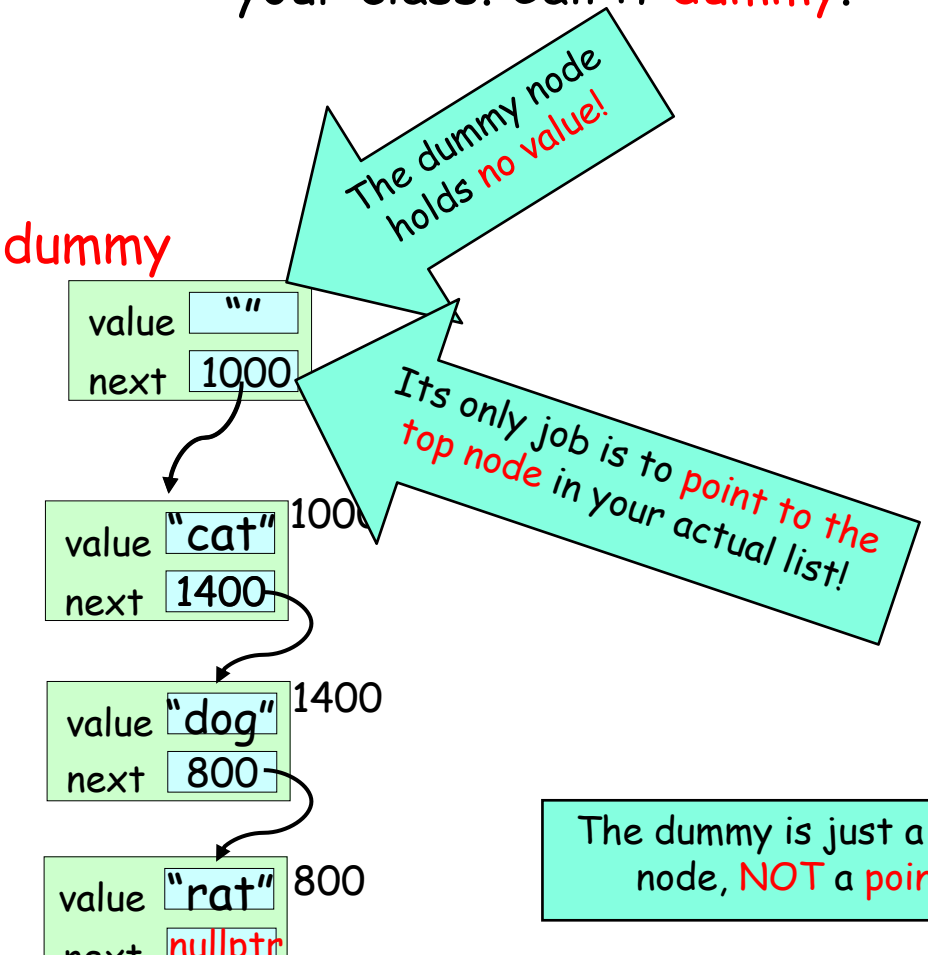
Head pointer!

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
    LinkedList() { … }
    void addToFront(string v) { … }
    void addToRear(string v) { … }
    void deleteItem(string v) { … }
    bool findItem(string v) { … }
    void printItems() { … }
    ~LinkedList() { … }

private:
    Node *head;
};
```

Linked Lists with a Dummy Node

Step #1:
Get rid of your head pointer!

Step #2:
Add a node member variable to your class. Call it dummy.



The dummy is just a regular node, NOT a pointer!

```
struct Node
{
    string value;
    Node   *next;
};
class LinkedList
{
public:
    LinkedList() { … }
    void addToFront(string v) { … }
    void addToRear(string v) { … }
    void deleteItem(string v) { … }
    bool findItem(string v) { … }
    void printItems() { … }
    ~LinkedList() { … }

private:
    Node dummy;
};
```

Linked Lists with a Dummy Node

Step #1:
Get rid of your head pointer!

Step #2:
Add a node member variable to your class. Call it dummy.

Step #3:
Update your member functions to use the dummy node.

(Generally, this involves removing code that deals with the head pointer from your member functions!)

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
   LinkedList()
   {
      Set dummy.next to nullptr
      Initialize node's value
   }

private:
   Node dummy;
};
```

Linked Lists with a Dummy Node

Step #1:
Get rid of your head pointer!

Step #2:
Add a node member variable to your class. Call it dummy.

Step #3:
Update your member functions to use the dummy node.

(Generally, this involves removing code that deals with the head pointer from your member functions!)

Why does this work?
Since every node in your list is GUARANTEED to have a parent node (either the dummy or another valid node), it let's you treat every node the same way and eliminate special-case code for dealing with the head pointer.

```
struct Node
{
    string value;
    Node  *next;
};
class LinkedList
{
public:
   void deleteItem(string v)
   {
     If the list is empty, return!
     If v is in the first node
        Update the head pointer to
           point to the second node
        Delete the first node
     Else
        Find the node above the one
          you want to delete
        Relink above node to the node
          below the to-delete node
        Delete the target node
   }

private:
   Node dummy;
};
```