

EECS 280

Discussion #12

Week of March 31

Outline

- **Administrivia**
- Deep Copies
- Linked Lists

Administrivia

- Project 4
 - Should be graded early next week
 - No release of grading test cases
- Project 5
 - Due April 15th @ 11:59 PM
- Discussion: 1 more content, then review

Outline

- Administrivia
- **Deep Copies**
 - Shallow Copies
 - Enter Pointers...
 - Resolution
- Linked Lists

Shallow Copies

- Consider instances when we “copy” structures/objects
 - Pass by value
 - Assignment
- By default, values of member variables are copied
 - Isn't this what we want?

A Shallow Copy

```
class MyStuff {  
    public:  
        int foo;  
        char bar;  
};
```

```
MyStuff a;  
MyStuff b;  
a.foo = 5;  
a.bar = 'c';  
b = a;
```


A Shallow Copy

```
class MyStuff {  
    public:  
        int foo;  
        char bar;  
};
```

```
MyStuff a;  
MyStuff b;  
a.foo = 5;  
a.bar = 'c';  
b = a;
```

a
foo:5
bar:c

b
foo:5
bar:c

Enter Pointers...

```
class MyStuff2 {  
    public:  
        int *foo;  
};
```

```
MyStuff2 a;  
MyStuff2 b;  
a.foo = new int(5);  
b = a;
```


Enter Pointers...

```
class MyStuff2 {  
    public:  
        int *foo;  
};
```

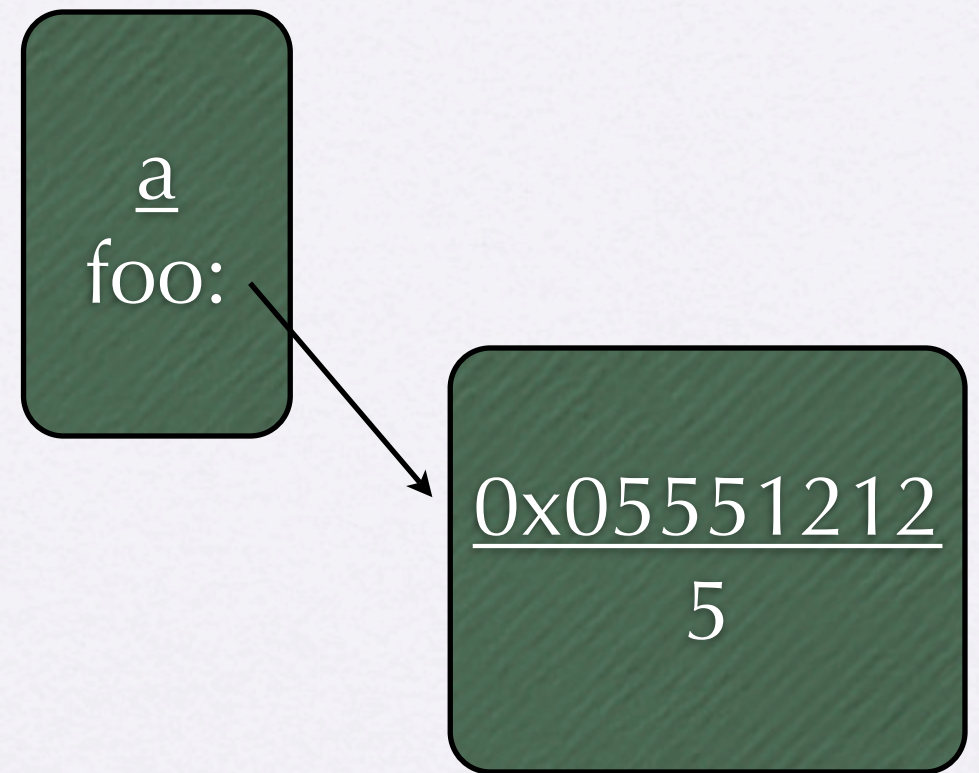


```
MyStuff2 a;  
MyStuff2 b;  
a.foo = new int(5);  
b = a;
```


Enter Pointers...

```
class MyStuff2 {  
    public:  
        int *foo;  
};
```

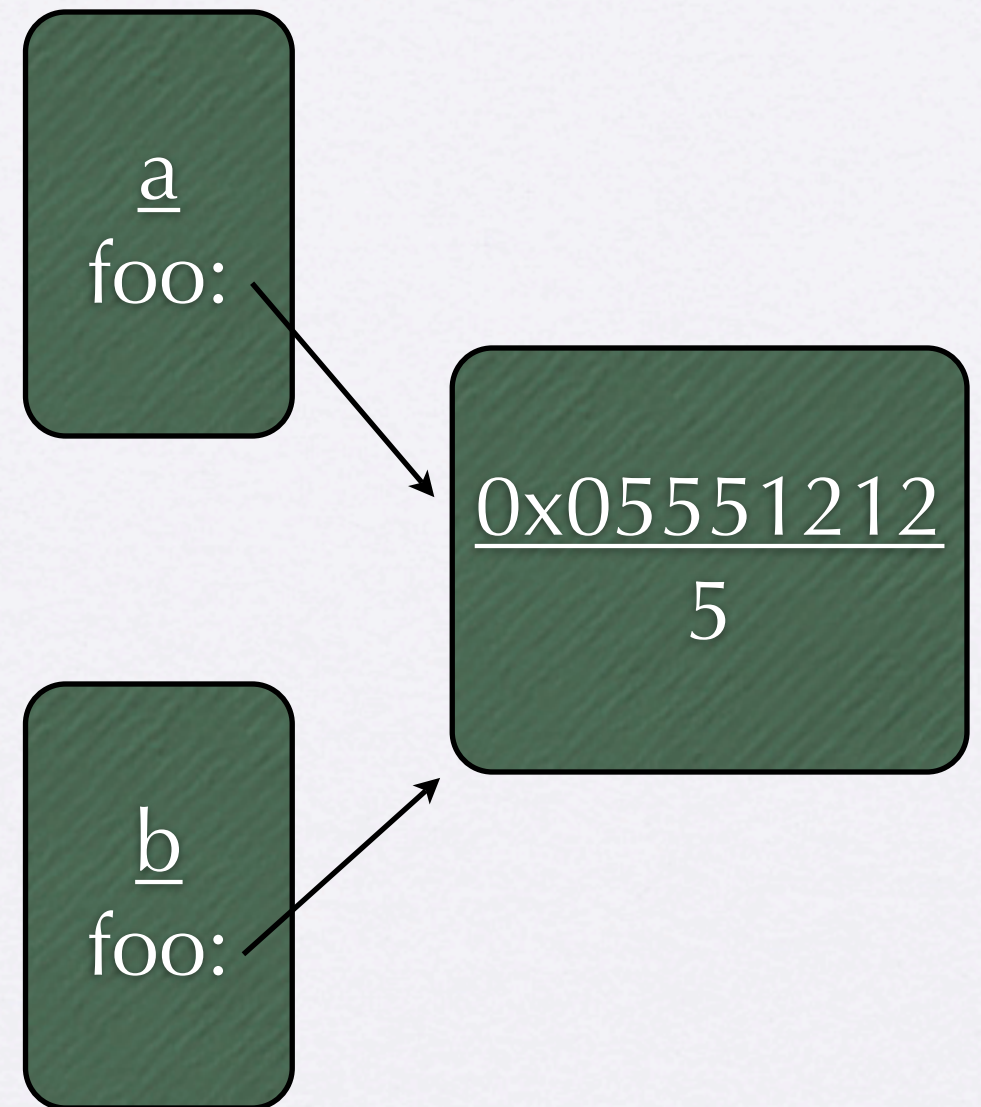
```
MyStuff2 a;  
MyStuff2 b;  
a.foo = new int(5);  
b = a;
```



Enter Pointers...

```
class MyStuff2 {  
    public:  
        int *foo;  
};
```

```
MyStuff2 a;  
MyStuff2 b;  
a.foo = new int(5);  
b = a;
```

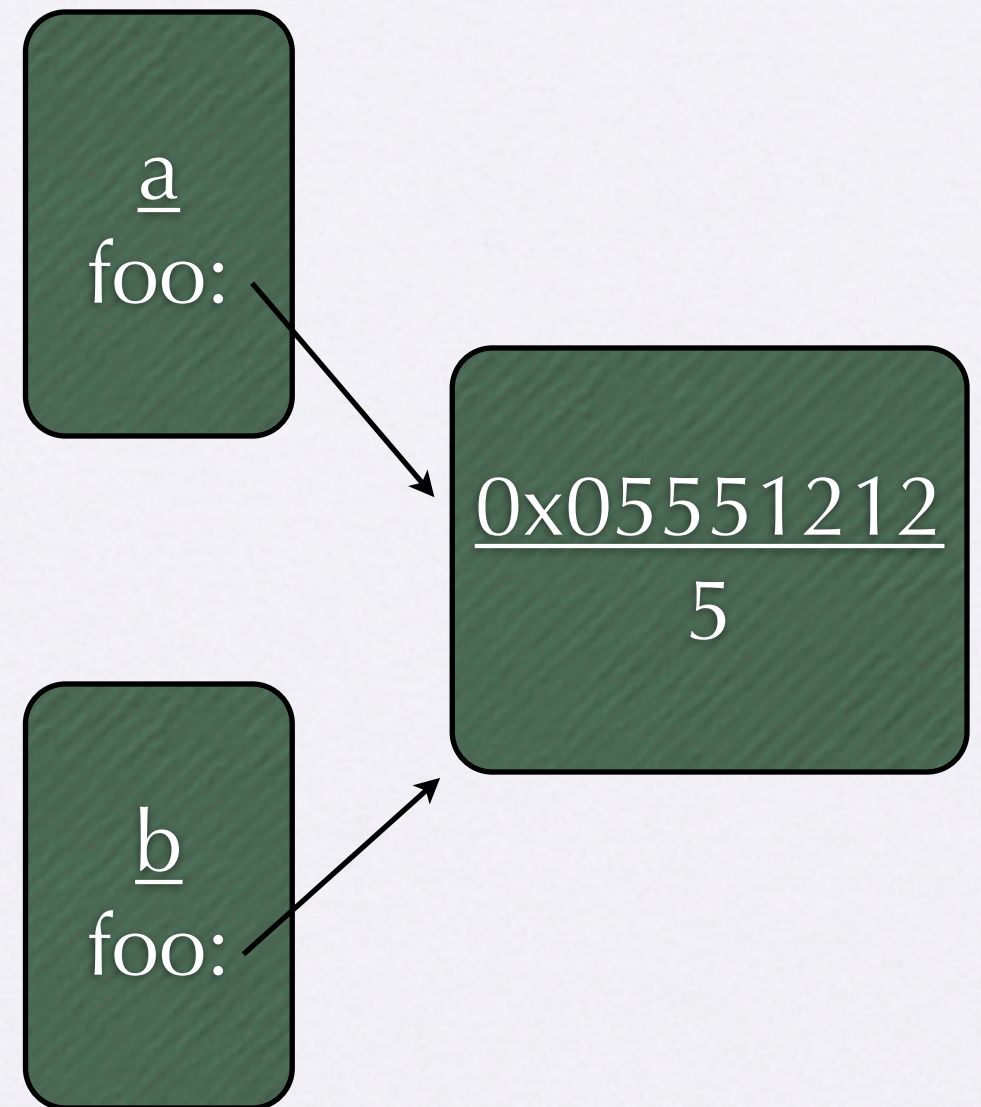


Enter Pointers...

Object “a” no longer owns the memory address stored in a.foo. It shares this location with “b.”

What happens if b changes the value at b->foo?

What happens if b deallocates b.foo (delete b.foo)?



Resolution

- In order to maintain ownership, we need to perform a deep copy
 - Allocate new memory for any dynamic elements in the source object/structure
- To make sure C++ performs a deep copy, we make two [related] changes
 - Add a copy constructor
 - Overload the assignment operator
- See notes for a fully implemented example

Outline

- Administrivia
- Deep Copies
- **Linked Lists**
 - Motivation
 - A View from *Memory*
 - Insert at Head
 - Linking Students

Motivation

Motivation

- Linked lists form the basis for managing data that changes size over time

Motivation

- Linked lists form the basis for managing data that changes size over time
- Currently, we have a one-to-one correspondence between variables and data

Motivation

- Linked lists form the basis for managing data that changes size over time
- Currently, we have a one-to-one correspondence between variables and data
 - But we cannot change the number of variables in our source code, nor can we have infinite numbers of variables at our disposal!

Motivation

- Linked lists form the basis for managing data that changes size over time
- Currently, we have a one-to-one correspondence between variables and data
 - But we cannot change the number of variables in our source code, nor can we have infinite numbers of variables at our disposal!
- Problem: how to keep track of a changing, potentially large amount of data with a finite [hopefully small] number of variables

Linked Lists

Linked Lists

- Problem: how to keep track of a changing, potentially large amount of data with a finite [hopefully small] number of variables

Linked Lists

- Problem: how to keep track of a changing, potentially large amount of data with a finite [hopefully small] number of variables
- Solution: use dynamic memory to store user data as well as [meta-]information about how we connect to more dynamic memory

Linked Lists

- Problem: how to keep track of a changing, potentially large amount of data with a finite [hopefully small] number of variables
- Solution: use dynamic memory to store user data as well as [meta-]information about how we connect to more dynamic memory
- A linked list is a connected set of “nodes,” where each node contains a unit of information as well as pointers/addresses to other nodes

A Node

```
struct node {  
    int data;  
    node *next;  
}
```


Some Nodes

```
struct node {  
    int data;  
    node *next;  
}
```

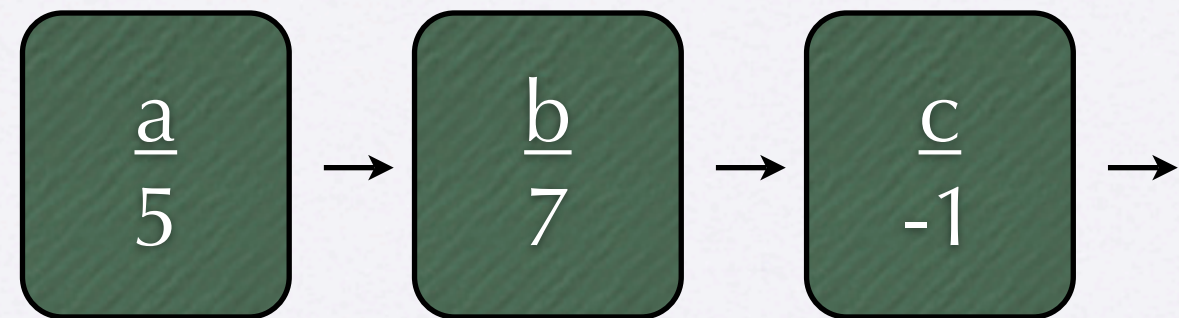
```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

```
a->data = 5;  
b->data = 7;  
c->data = -1;
```

```
a->next = b;  
b->next = c;  
...
```


Some Nodes

```
struct node {  
    int data;  
    node *next;  
}
```



```
node *a = new node;  
node *b = new node;  
node *c = new node;  
a->data = 5;  
b->data = 7;  
c->data = -1;
```

```
a->next = b;  
b->next = c;  
...
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	?	?	?	?	?	?	?	?	?	?	?
variable:												

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:												

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	?	?	?	?	?	?	?	?	?	?	?
variable:												

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:												

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

A node pointer requires only one memory unit.
An actual node requires two memory units.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	?	?	?	?	?	?	?	?	?	?	?
variable:	a											

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:												

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

A node pointer requires only one memory unit.
An actual node requires two memory units.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	?	?	?	?	?	?	?
variable:	a											

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:	X	X										

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

A node pointer requires only one memory unit.
An actual node requires two memory units.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	?	?
variable:		a					b					

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:	X	X					X	X				

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

A node pointer requires only one memory unit.
An actual node requires two memory units.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:		a				b					c	

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:	X	X		X	X		X	X				

```
node *a = new node;  
node *b = new node;  
node *c = new node;
```

A node pointer requires only one memory unit.
An actual node requires two memory units.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a			b				c				

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:	X	X		X	X		X	X				

a->data = 5;

b->data = 7;

c->data = -1;

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a			b				c				

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	?	?	?	?	?	?	?	?	?	?	?
allocated:	X	X		X	X		X	X				

```
a->data = 5;  
b->data = 7;  
c->data = -1;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a			b			c					

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	?	?	?	?	?	7	?	?	?	?	?
allocated:	X	X		X	X		X	X				

`a->data = 5;`

`b->data = 7;`

`c->data = -1;`

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a			b			c					

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	?	?	-1	?	?	7	?	?	?	?	?
allocated:	X	X		X	X		X	X				

`a->data = 5;`

`b->data = 7;`

`c->data = -1;`

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a				b				c			

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	?	?	-1	?	?	7	?	?	?	?	?
allocated:	x	x		x	x		x	x				

a->next = b;

b->next = c;

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a				b				c			

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	?	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

a->next = b;

b->next = c;

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:		a				b				c		

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	?	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

```
a->next = b;  
b->next = c;
```

Now given only a (the address) we can get to all the nodes!

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a				b				c			

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	?	?	7	103	?	?	?	?
allocated:	x	x		x	x		x	x				

But how do we know
when to stop?

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:		a				b				c		

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	0	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

```
c->next = NULL;
```

But how do we know
when to stop?

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:		a				b				c		

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	0	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

```
c->next = NULL;
```

Since NULL is never a valid object address, we use it as a flag to stop.

A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:	a				b				c			

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	0	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

```
// consider  
a->next->next->data
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	106	?	?	?	?	103	?
variable:		a				b				c		

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	5	106	?	-1	0	?	7	103	?	?	?	?
allocated:	X	X		X	X		X	X				

```
// consider  
a->next->next->data
```

$*(*(100 + 1) + 1)$
-1

Insert at Head

```
node *head = NULL, newbie;
```

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```

```
...
```


Insert at Head

```
node *head = NULL, newbie;
```

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;  
...
```


Using this form of insertion (insert at head), we can create a linked list of arbitrary length with two variables!

Insert at Head

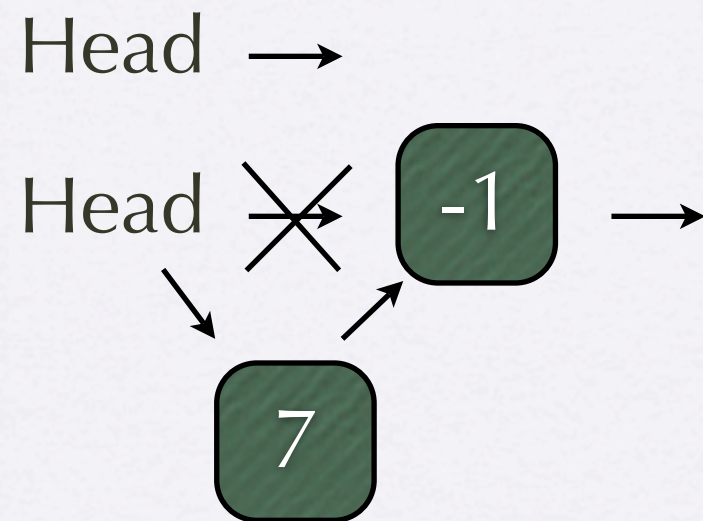
Head →

Insert at Head

Head →

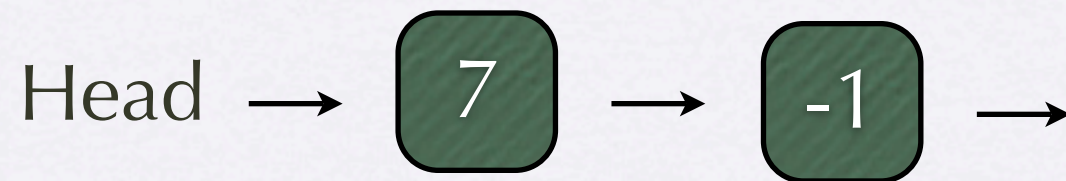
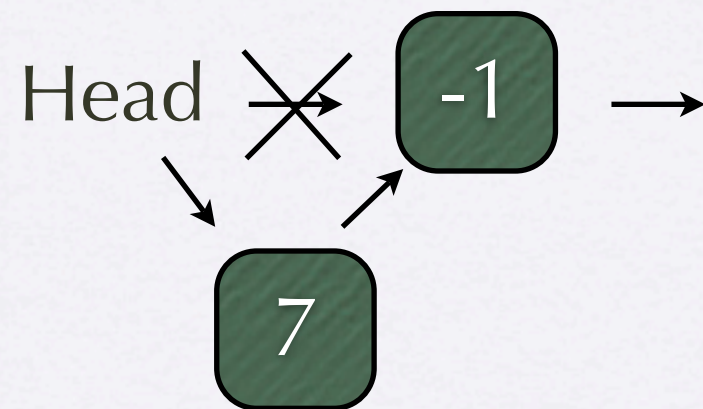
Head →  →

Insert at Head



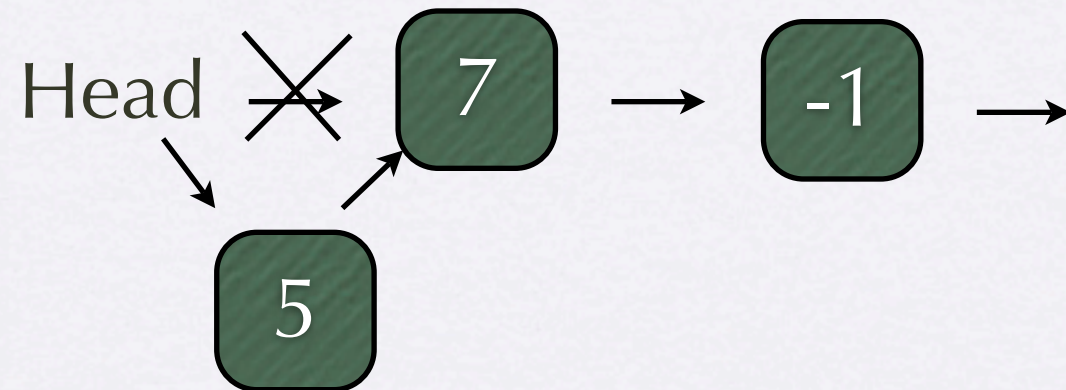
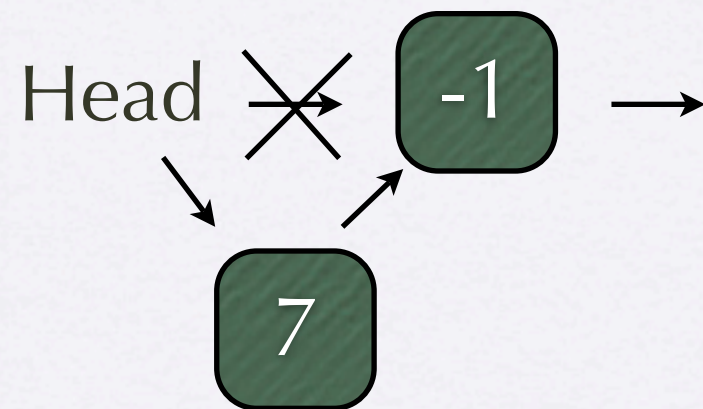
Insert at Head

Head →



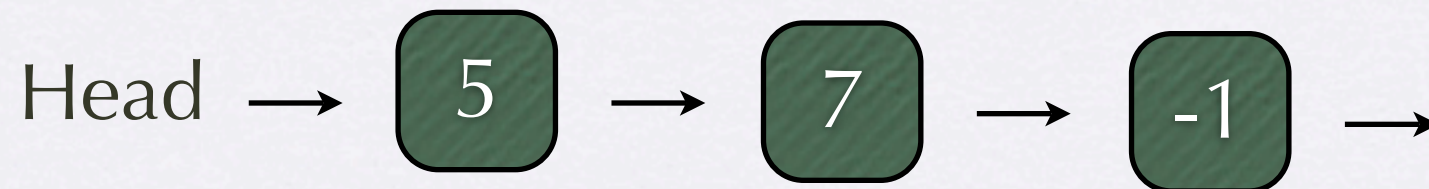
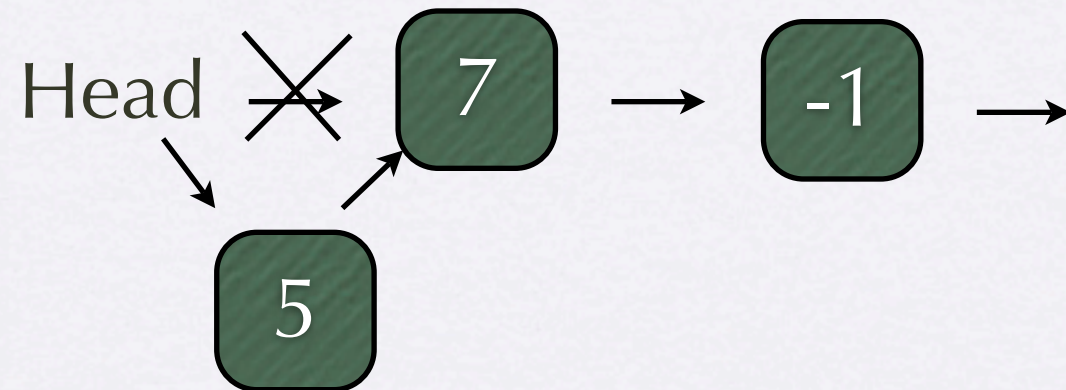
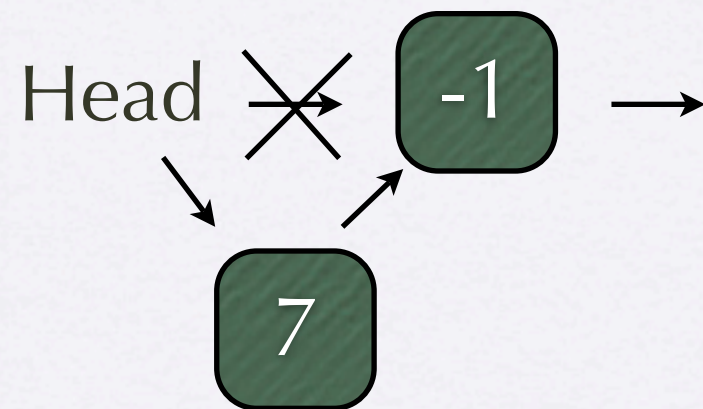
Insert at Head

Head →



Insert at Head

Head →



A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	0	?	?	?	?	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:												

```
// abbreviated as "h"  
node *head = NULL;
```

```
// abbreviated as "n"  
node *newbie
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	0	?	?	?	?	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:												

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	0	?	?	?	100	?	?	?	?	?	?
variable:		h					n					

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	?	?	?	?	?	?	?	?	?	?	?	?
allocated:	x	x										

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	0	?	?	?	100	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	?	?	?	?	?	?	?	?	?	?	?
allocated:	x	x										

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	0	?	?	?	100	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	?	?	?	?	?	?	?
allocated:	X	X										

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	100	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	?	?	?	?	?	?	?
allocated:	x	x										

```
newbie = new node;  
newbie->data = -1;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	100	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	?	?	?	?	?	?	?
allocated:	X	X										

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	?	?	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	7	?	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	100	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	7	100	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	105	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	7	100	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```


A View from Memory

Stack

address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	105	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	7	100	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```

We have developed a simple, repeatable set of steps to add to our list with only two variables...

A View from Memory

Stack

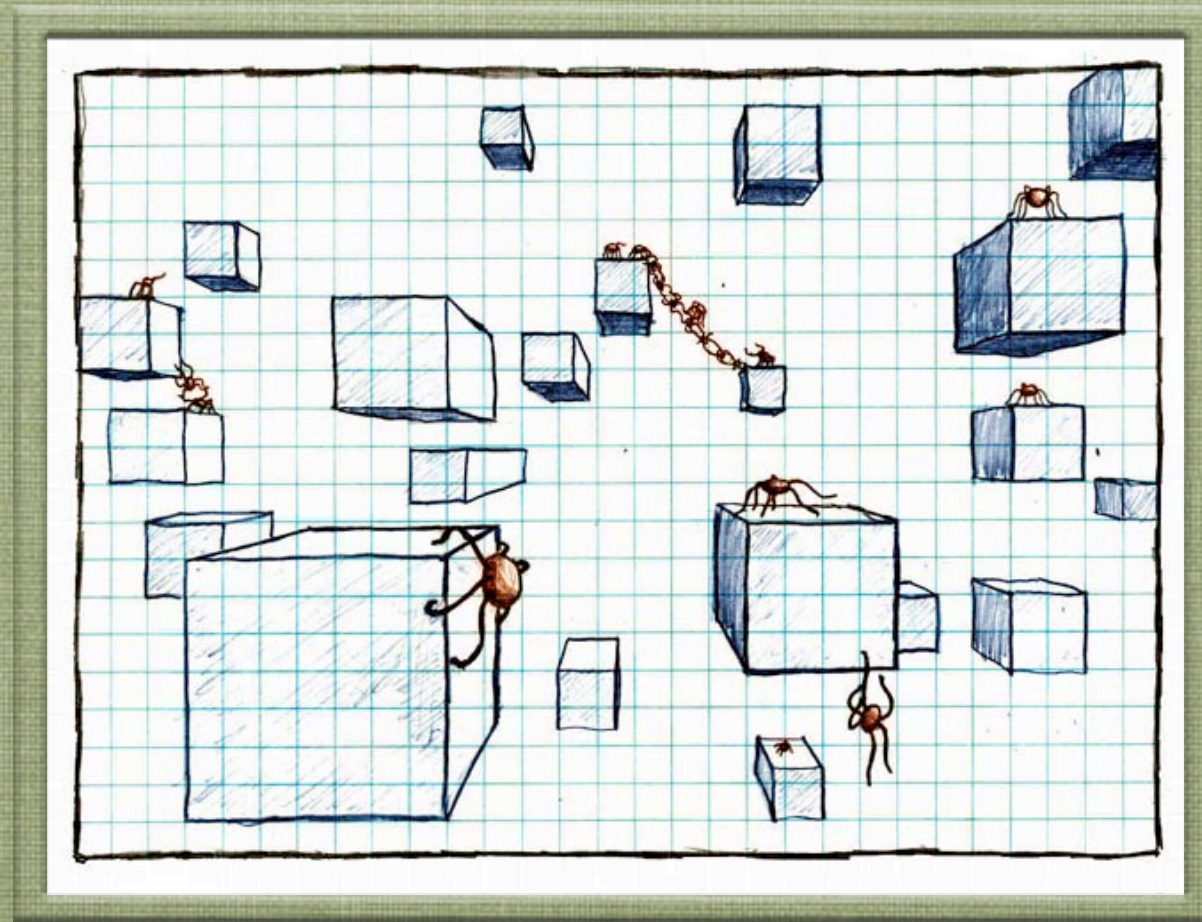
address:	0	1	2	3	4	5	6	7	8	9	10	11
contents:	?	105	?	?	?	105	?	?	?	?	?	?
variable:	h					n						

Heap

address:	100	101	102	103	104	105	106	107	108	109	110	111
contents:	-1	0	?	?	?	7	100	?	?	?	?	?
allocated:	X	X				X	X					

```
newbie = new node;  
newbie->data = 7;  
newbie->next = head;  
head = newbie;
```

what about deletion...?



Linking Students