Linked Lists

Lecture 10

A Preview of Data Structures

- Currently we use an array or vector to organize a large number of variables
- Issue: when an item needs to be added or removed from the middle, lots of copying/ moving is necessary
- A *linked list* is a **data structure** that does not suffer this inefficiency
  - Though it has its own tradeoffs

A Linked List

- A linked list is composed of a set of **nodes**, where each node is a class/ structure with two sets of information
  - Data: what the node stores
  - Pointer(s): what the node is connected to in the list

- By using dynamic memory, pointers, and classes, we can grow/shrink/manipulate this list of connected nodes

A Node [in Code]

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

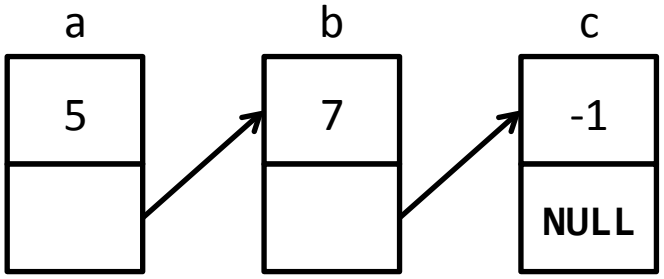
Some Nodes

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

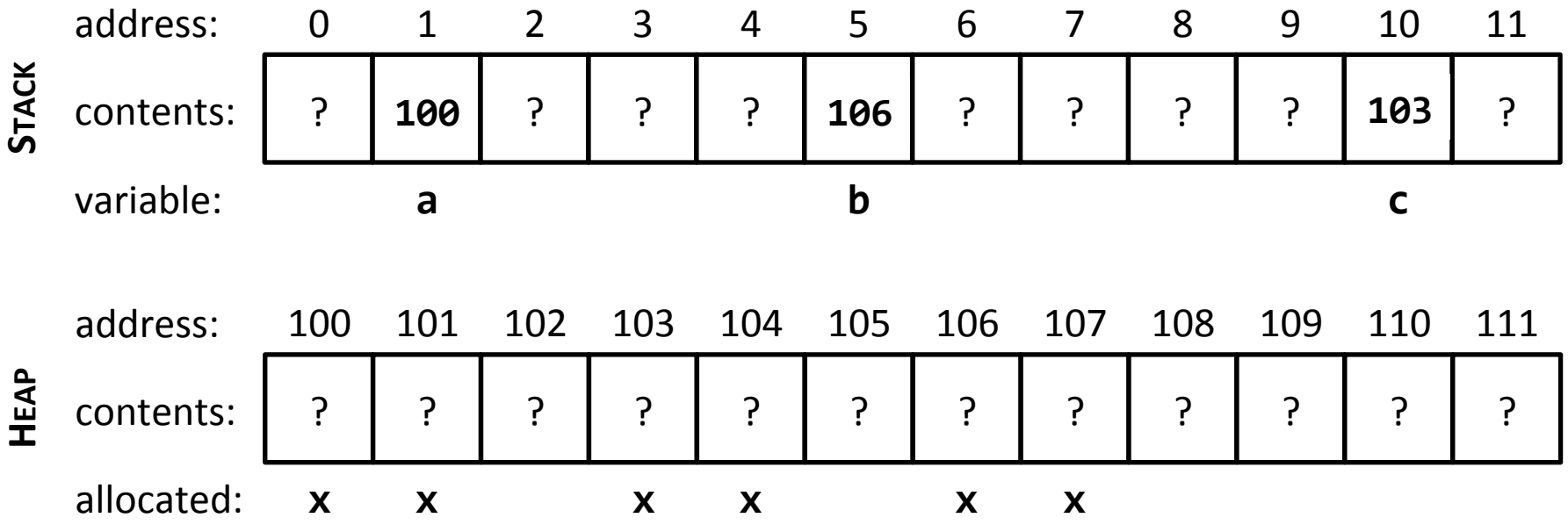
```
node *a = new node;
node *b = new node;
node *c = new node;

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

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



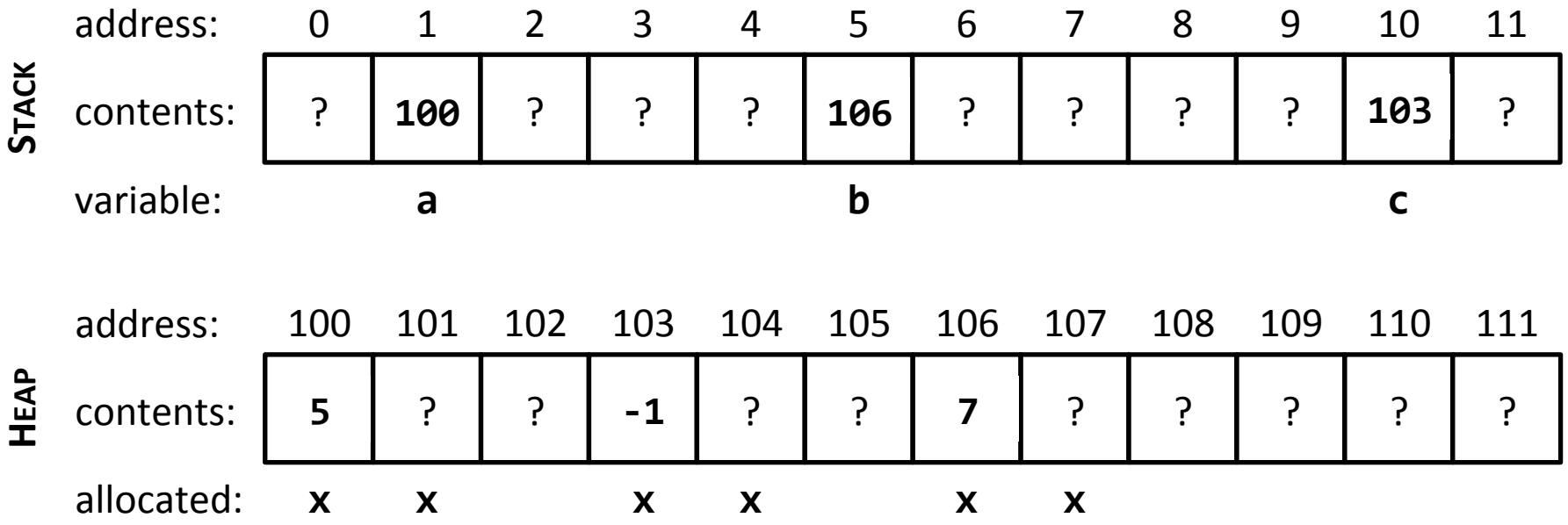
A View from Memory




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

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

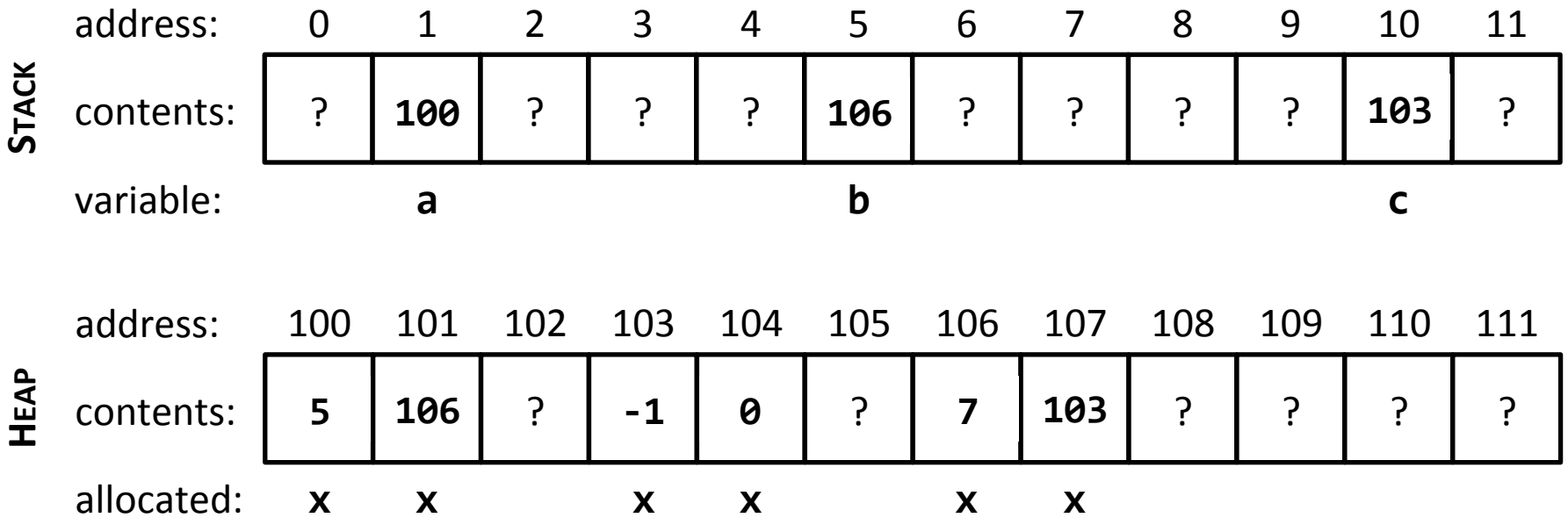
A View from Memory




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

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

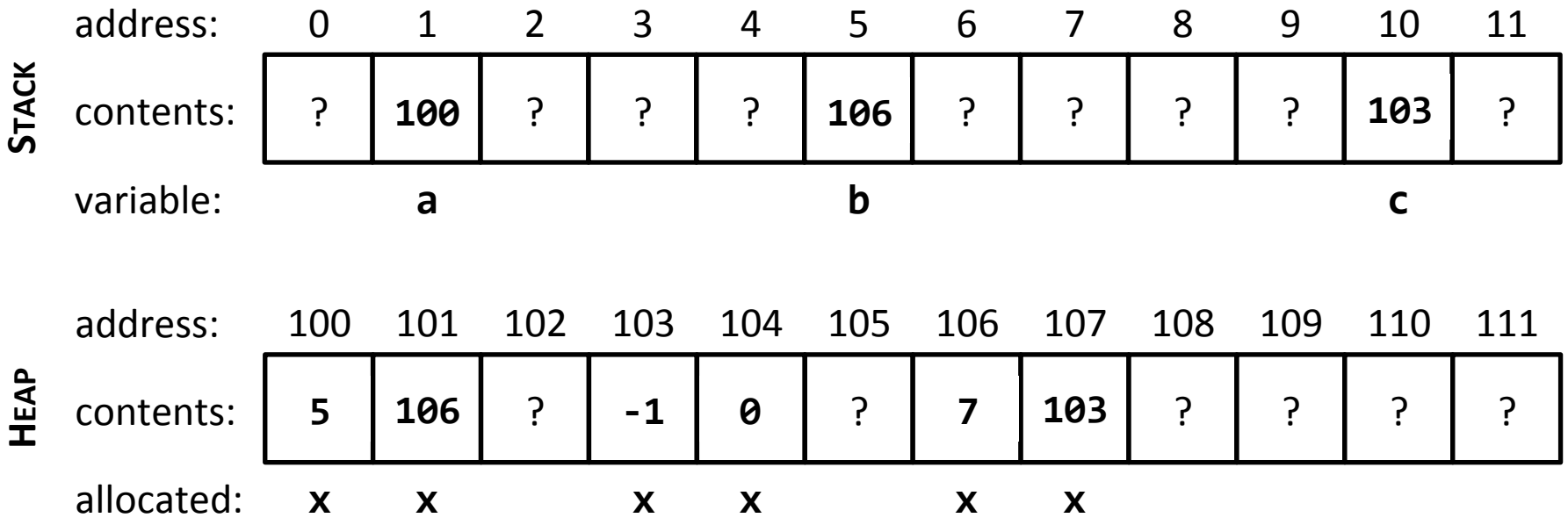
A View from Memory




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

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

Exercise




```
cout << a->next->next->data;
// -1
```

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

Linked List of Nodes

- A simple linked list has a single variable that points to the front of the list, typically called the **head** of the list

```
node* head;
```

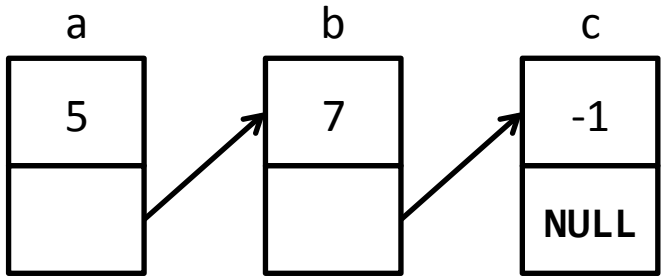
- From there, every node points to the next node, until the last points to **NULL**
  - At the start, an empty list has head=**NULL**

- Now start writing useful functions for the list!

Exercise

Write a function named **`printList`** that takes as an argument a node pointer and prints the list to the screen.

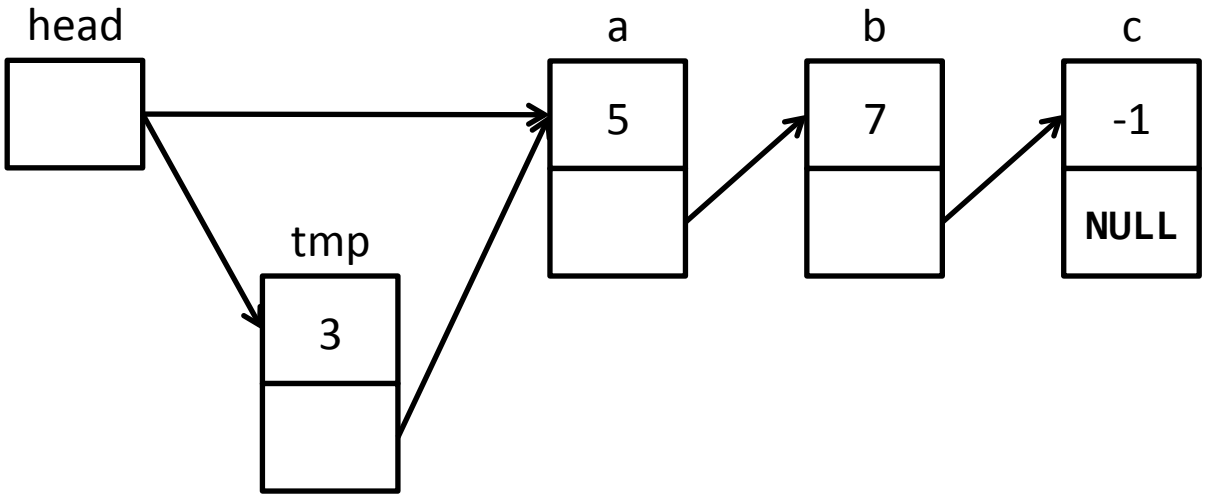
```
struct node
{
    int data;
    node* next;
};
```



Answer

```
void printList(node* head)
{
   cout << "[";
   for ( node* n=head; n!=NULL; n=n->next )
   {
      cout << " " << n->data;
   }
   cout << " ]" << endl;
}
```

Add to Front



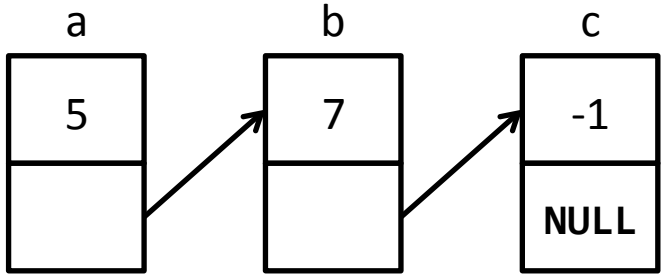
1. Create new node, tmp
   - Set data
2. Set tmp->next = head
3. Set head = tmp

```
list.addToFront( 3 );
```

Exercise

Write an **`addToFront`** function that takes as arguments a reference to a node pointer, as well as an integer, and adds a new node to the front of the list with the integer as its data.

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



Answer

```
void addToFront(node*& head, int val)
{
   node *tmp = new node();
   tmp->data = val;
   tmp->next = head;
   head = tmp;
}
```

Watch Out for Leaks!

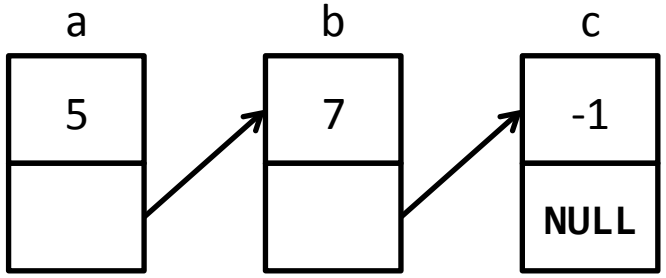
When writing linked list code, it becomes very easy to lose track of your list via bad/ out-of-order pointer assignment

```
void addToBad(node*& head, int val)
{
  head = new node();
  head->data = val;
  head->next = ?
}
```

Exercise

Write the function **`nodeValueExists`** that takes the head of the list and an integer as arguments and returns true if the integer value is found in the list.

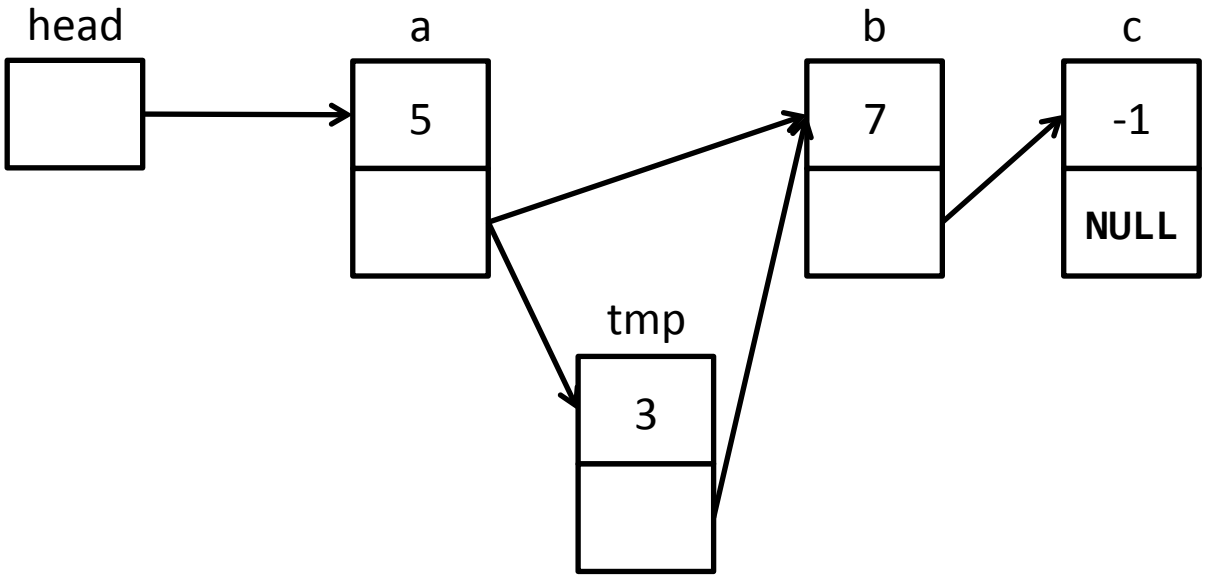
```
struct node
{
    int data;
    node* next;
};
```



Answer

```
bool nodeValueExists(node* head, int val)
{
   for ( node* n=head; n!=NULL; n=n->next )
   {
      if ( n->data == val )
         return true;
   }
   return false;
}
```

Add After



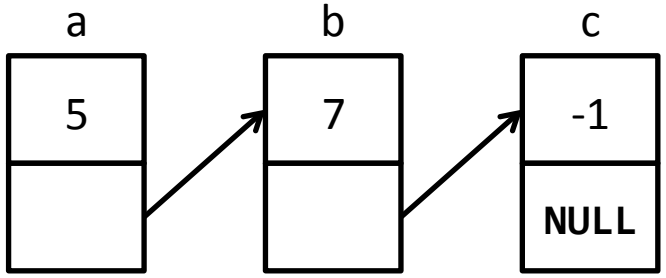
1. Create new node, tmp
   - Set data
2. Set tmp->next = a->next
   - Order matters!
3. Set a=tmp

```
list.addAfter( a, 3 );
```

Exercise

Write the **`addAfter`** function to insert a node in the middle of an existing list.

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



Answer

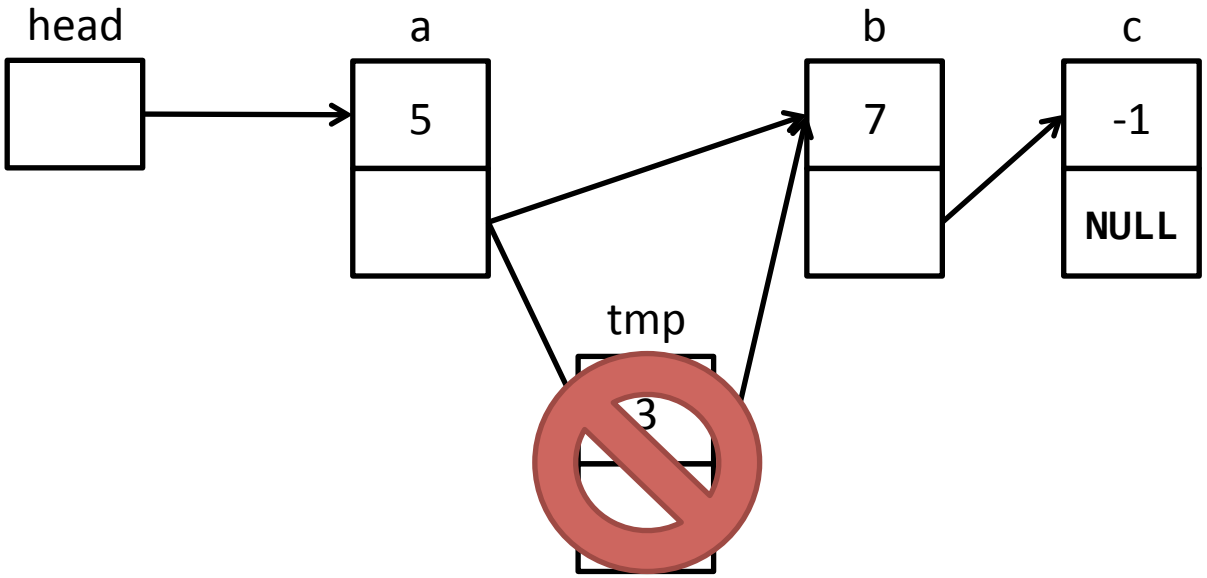
```
void addAfter(node* afterMe, int val)
{
   node* tmp = new node();
   tmp->data = val;
   tmp->next = afterMe->next;
   afterMe->next = tmp;
}
```

A Note on Efficiency

- Note that no matter how long the list is, or where we insert, it only requires two pointer assignments

- This is a dramatic difference from arrays!
  - But how long does it take to get to the n^{th} node? vs. array?

Remove After



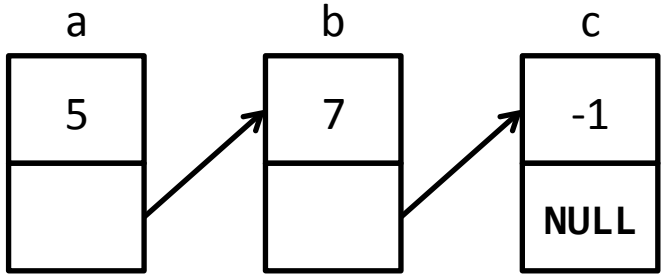

```
list.removeAfter( a );
```

1. tmp = a->next
2. Set a->next=tmp->next
3. Deallocate tmp

Exercise

Write the **`removeAfter`** function to remove a node from a list.

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



Answer

```
void removeAfter(node* afterMe)
{
  node* tmp = afterMe->next;
  if ( tmp != NULL )
  {
    afterMe->next = tmp->next;
    delete tmp;
  }
}
```

Wrap Up

- A linked list is a data structure for efficiently growing/shrinking a sequence of items
- A linked list is composed of a set of linked nodes, each having data and [at least one] pointer
- Our prior work on classes, dynamic memory management, and pointers allow us to implement useful functions on linked lists
  - Next lecture we will use the list to make even more complex structures!