Linked Lists

Phil Tayco

Slide version 1.6

Updated Sep 22, 2021

Linked Lists

Array pros and cons

- Arrays have many benefits:
  - Direct access to any element at any time
  - "Easier" to code
  - Can be used to apply binary search
- Like all structures, they have cons:
  - Memory is static. It must be reserved in advance and can lead to unused space
  - Does not conveniently allow for additional memory use. If the capacity is underestimated, getting more memory is an expensive procedure
  - Element shifts are O(n)
- When looking at new data structures, we start with addressing the cons of the previous ones
- Here, can we have a data structure that is not static, conveniently handles additional memory needs and avoids O(n) "shifting"?

Linked Lists

Memory References (a.k.a. Pointers)

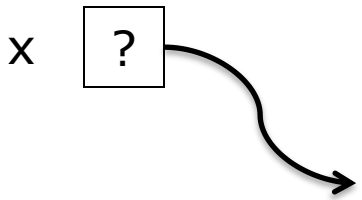
- Before diving into Linked Lists, we need to make sure we understand some other concepts
- By definition, a pointer is a variable that contains a memory address
- The address is the location where the value of a variable is stored
- When pointers are defined and memory is allocated, that space used by “dereferencing” the pointer (the pointer is a referred to as a “reference”)
- Modern programming languages have “no pointers”, but in fact they are handling a lot of the pointer process automatically for you, especially when applied to abstract data types (a.k.a. classes and objects)
- This makes it important to understand how variables that are references to other places in memory that store data work

Linked Lists

Reference example in Java:

```
// Reference declaration – Create a reference
// variable called x that will point to a location
// in memory that will store a Student object

Student x;
```



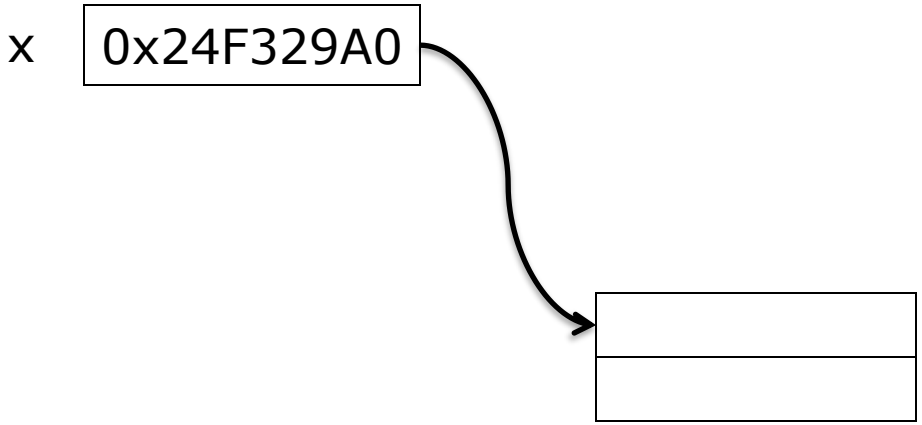

```
// Memory has not yet been reserved. x is only
// set up at this point to reference something
// that will represent Student data
```

Linked Lists

Reference example in Java:

```
// Memory allocation – this is creating the Student object
// in memory. x contains the location of the object
// (e.g. x is the reference or pointer)

x = new Student();
```

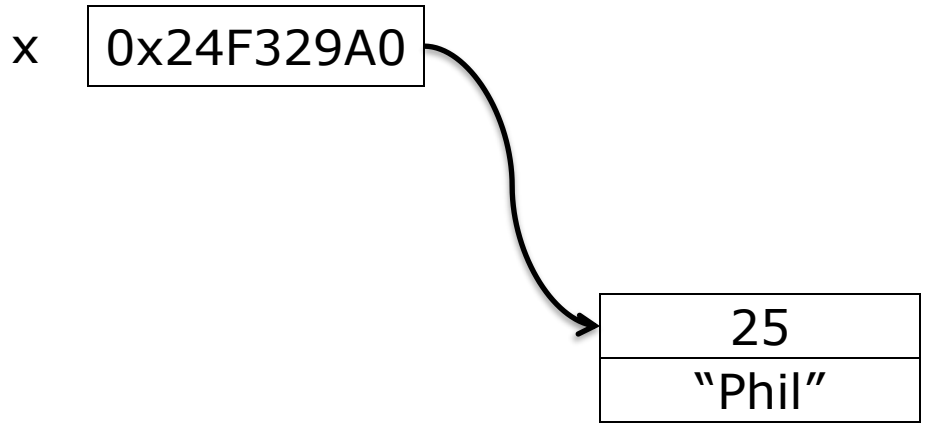


Linked Lists

Reference example in Java:

```
// Store 25 and "Phil" in the Student object referenced
// by x. Each line of code is "dereferencing" x by first
// going to the location in memory where the object
// resides and then store the value in the corresponding
// property

x.id = 25;
x.Name = "Phil"
```



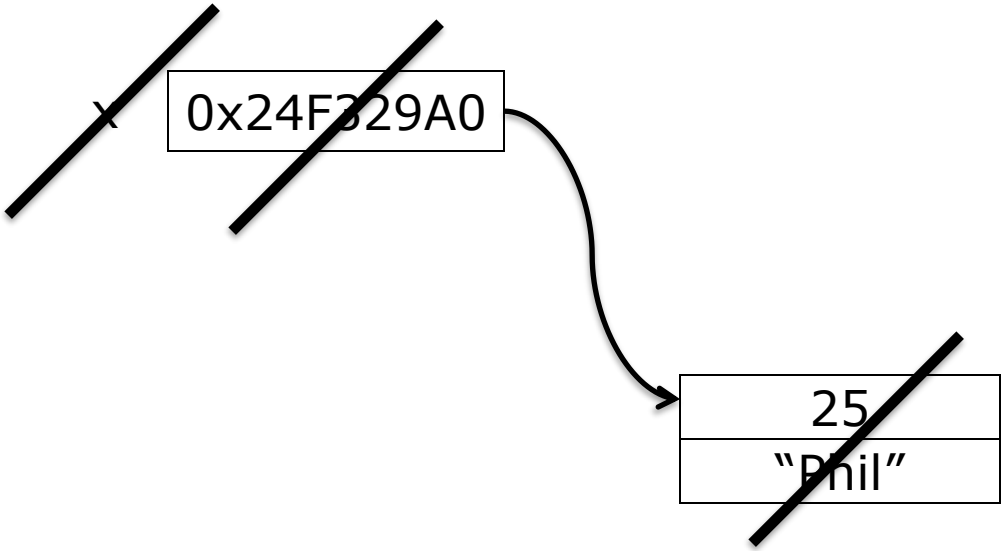
Linked Lists

Reference example in Java:

```
// Deallocate the memory when you are done. This frees the
// memory and the pointer variable.

x = null;

// The process is similar in C++, but uses a different
// syntax like “Student *x”, “x->id = 25;”, and “delete x”
```



Linked Lists

Classes and structures

- A class or structure is an abstract data type defined by the programmer
  - int x; is a primitive data type given to you by the compiler. It already knows what an integer is and expects x to store whole numbers
  - Student x; would be an abstract data type. The compiler does not know what a Student is and would need to be defined
  - Once a Student is defined, you can create any number of variables like x of that type to store different values. We call these variables "objects"
- We will use the term "class" for these abstract data types

Linked Lists

Class properties

- All classes are defined by their “attributes” or “properties”
- They model the concept the data type is named for and answers the question, “what does this class have?”
- A Student, for example, has a first name, last name, id, age, gender and a GPA. There are many other properties, so we limit it to what we want to capture

Linked Lists

Class code (Java)

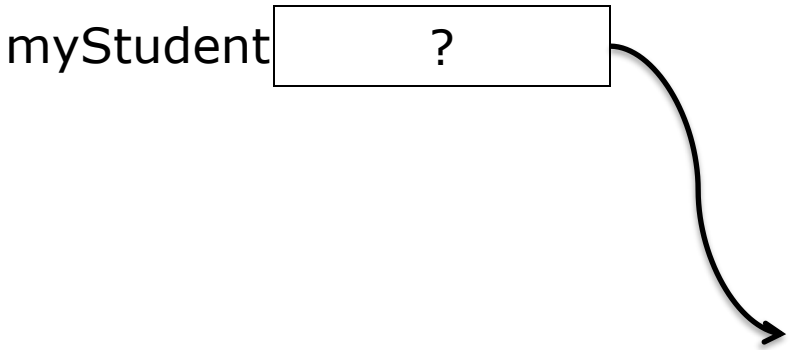
```
public class Student
{
   public String firstName;
   public String lastName;
   public int age;
   public double gpa;
   public char gender;
   public String id;
}

// Any variable that uses Student as a data type will have its
// own firstName, lastName, age, gpa, gender and id with values
// that must support these property data types
```

Linked Lists

Class code (Java)

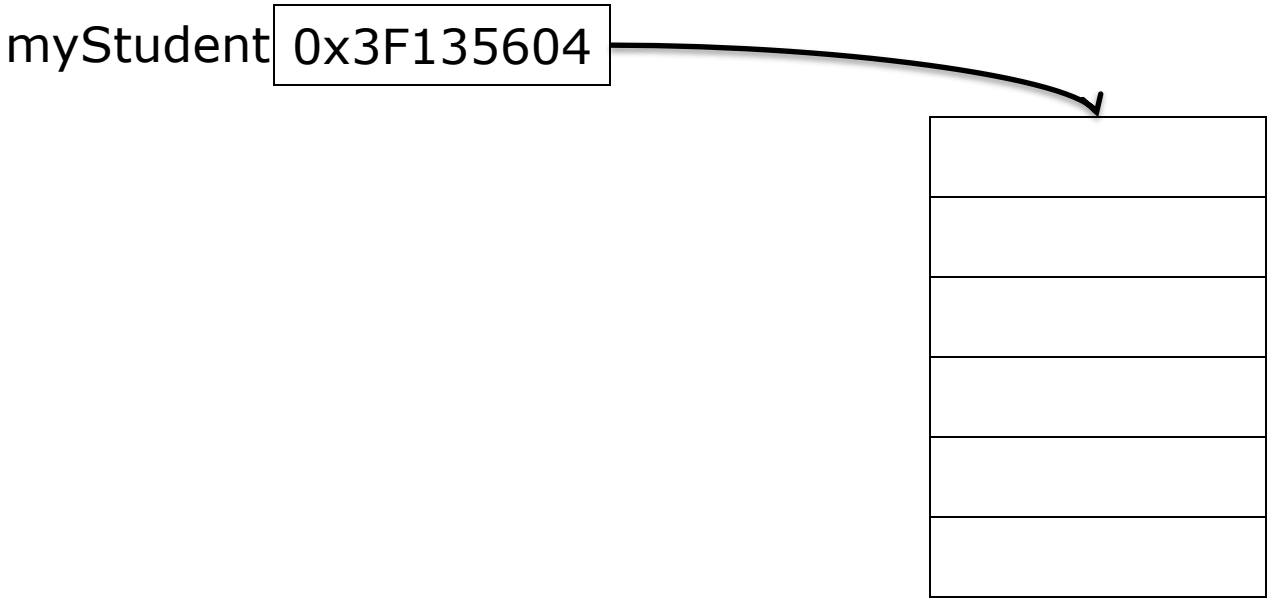
```
Student myStudent;     // Only creates the pointer reference
                       // No memory allocated yet…
```



Linked Lists

Class code (Java)

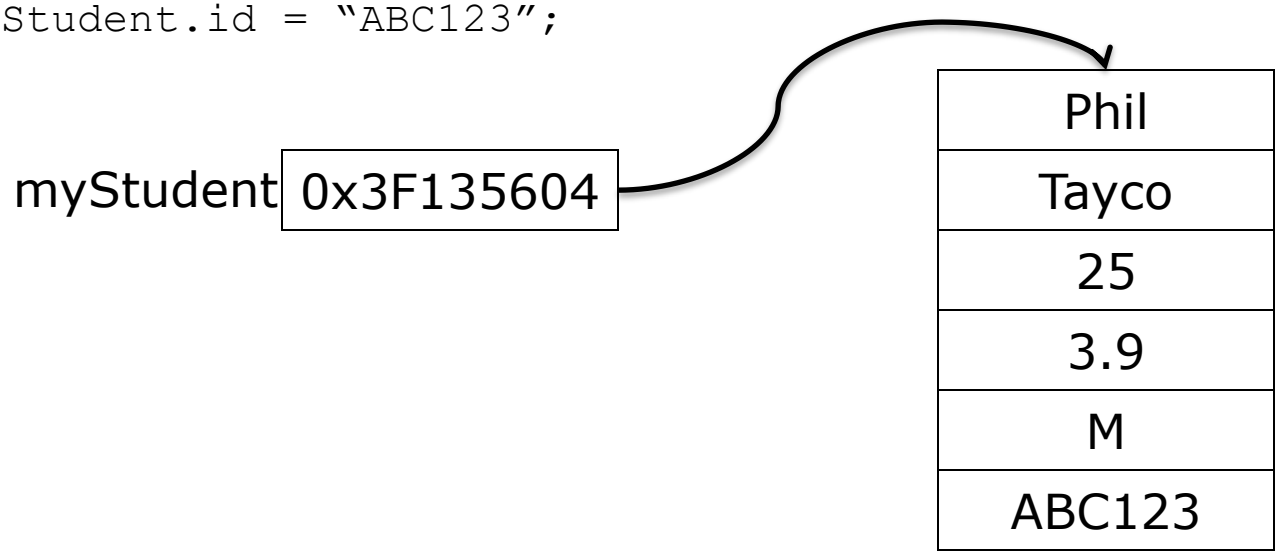
```
myStudent = new Student();
// Memory is now allocated with enough space to store what is
// defined in a Student. myStudent has its own instance of
// Student information
```



Linked Lists

Class code (Java)

```
myStudent.firstName = "Phil";
myStudent.lastName = "Tayco";
myStudent.age = 25;
myStudent.gpa = 3.9;
myStudent.gender = 'M';
myStudent.id = "ABC123";
```



Linked Lists

Code observations

- Student objects are “instantiated” with the “new Student()” line. From there, the object properties are treated like normal variables
- myStudent is one instance of type Student with its own set of values for each of its properties. Similar code can be used with different objects
- Abstract data types are treated like normal data types
  - You can create an array of Students
  - You can use a class as a property of another class (known as “composition”)
- The class definition is a template or a definition. Actual memory is not allocated until an object is instantiated
- Classes can also contain functions (called methods) which you will see used later. Designing and programming extensively with classes is beyond the scope of this course

Linked Lists

Dynamic memory allocation

- So what do we know to this point:
  - Abstract data types allow us to define our own types of structures
  - Class properties can use simple and complex data types within them
  - Pointers dynamically handle memory (we use it when we need it)
- We can combine the use of this knowledge to define a class that allows us to manage sets of data like an array does while also addressing the cons of that data structure
- First, we will define a class that contains properties for storing an element of data and handling memory references to other elements like it

Linked Lists

The Node class:

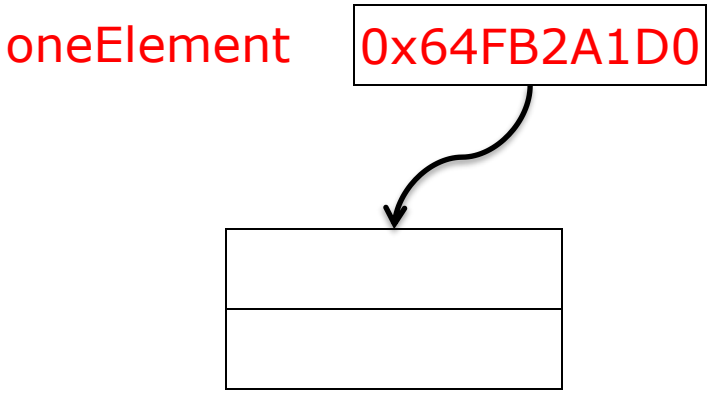
```
public class Node
{
   public int data;
   public Node next;
}

// Node is a data type the will represent one
// instance of data in a list like an element in
// an array

// “data” represents the actual value of the element
// “next” will refer to the next element in the list
```

Linked Lists

Using the Node class:




```
public class Node
{
   public int data;
   public Node next;
}
```

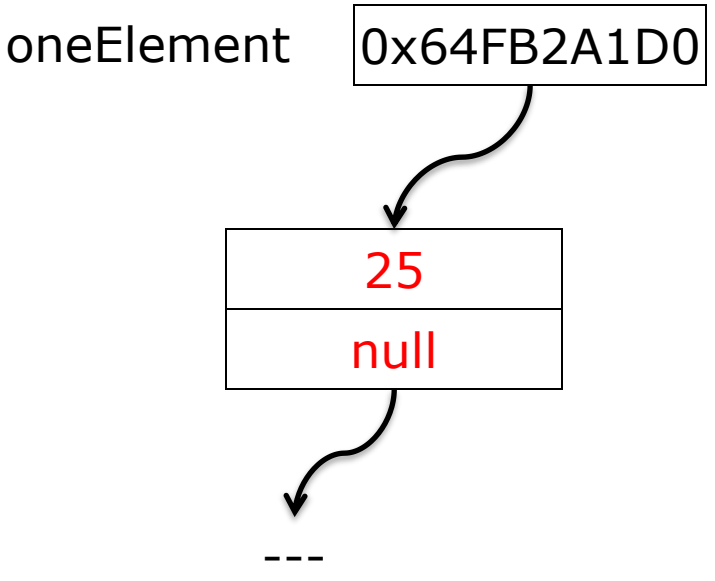
```
Node oneElement = new Node();

// Here, we create an instance of a Node called
// "oneElement". It currently references a location
// in memory where Node data will reside
// Currently, there is no data where oneElement
// is pointing to
```

Linked Lists

Using the Node class:

```
public class Node
{
   public int data;
   public Node next;
}
```



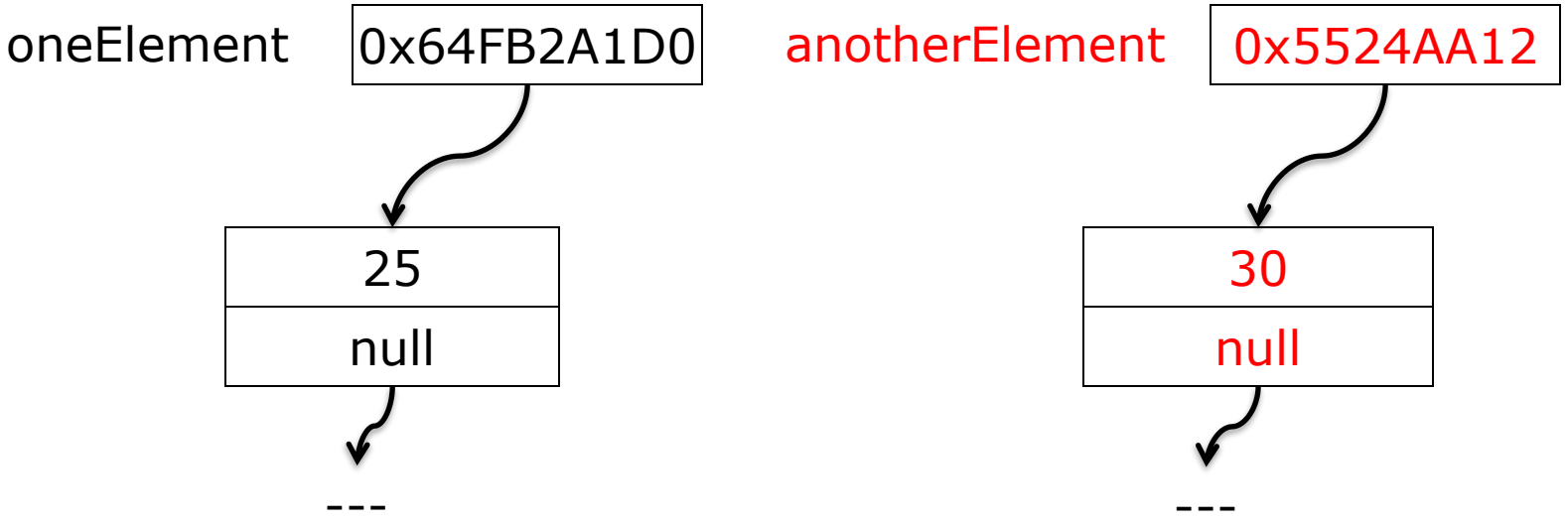

```
Node oneElement = new Node();
oneElement.data = 25;
oneElement.next = null;

// The properties of oneElement are now populated
// with data. Here, it represents storing a "25" and
// the "next" element does not exist (it's "null")
```

Linked Lists

Another node created indepedently:

```
Node anotherElement = new Node();
anotherElement.data = 30;
anotherElement.next = null;
```



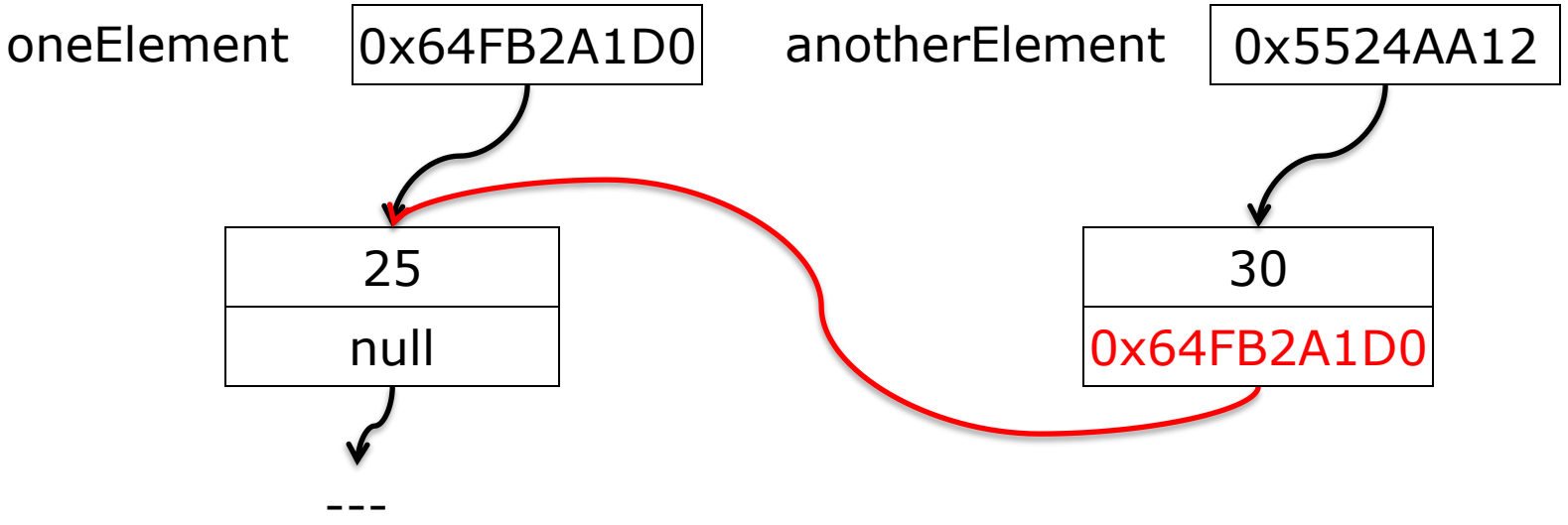
Linked Lists

- Notice the data type of the “next” property – it’s a Node
- This means that “next” can reference any other object that is also a Node
- In these examples, we created 2 node objects – oneElement and anotherElement
- Because these are both “Node” data types, the “next” pointer of either of these Node objects can be updated to point to the other one

Linked Lists

Another node linked to the first:

```
Node anotherElement = new Node();
anotherElement.data = 30;
anotherElement.next = oneElement;
```



Linked Lists

Node observations

- The Node class contains 2 essential properties
  - A "data" property which can really be any data type. We use int as an example, but it could be a String or another class
  - A "next" property whose data type is Node. This does not mean "next" is itself, but that "next" is a pointer to another object of type "Node"
- oneElement is a Node object with a data value of 25. The next Node object it is pointing to does not exist (its value is set to null)
- anotherElement is also a Node object with a data value of 30. Its next Node is pointing to oneElement
- The chain of Node objects is called a "Linked List"
  - We can create any number of Node objects at any time
  - We can add Node objects to the Linked List at any time making this data structure handle dynamic memory usage
- Here, "anotherElement" is considered to be the first element in this list of Nodes – we refer to the first Node of a Linked List as the "head" element

Linked Lists

Learning Linked Lists

- It is critical for your learning to draw pictures when understanding Linked List algorithms
- It is very easy to get lost with the references when coding changes to Node objects and "next" properties
- When studying Linked List code, don't just read the code, compile, and run it to see if it works – run through the execution of code yourself and step through each line of execution slowly
  - Using a debugger with example code and stepping line by line is very helpful
- Keep track of where objects and references are pointing (this is where drawing pictures help)

Linked Lists

- Now that we have a list data structure like an array, we can compare them and see what the pros and cons are
- We will use consistent notation (Big O) and look at the same 4 common functions (Search, Insert, Update and Delete)
- Questions to think about:
  - What do you think the pros and cons will be with Linked Lists?
  - How do you think they will compare to arrays?
  - Will order of data be a factor?

Linked Lists

The Linked List class:

```
public class Node
{
   public int data;
   public Node next;
}

public class LinkedList
{
   public Node head = null;

   public void insert(int value);
   public boolean update(int oldValue, int newValue);
   public boolean delete(int value);
   public Node search(int value);

   public void printList();
}
```

Linked Lists

Node and Linked List classes

- Think of the Node class as the definition of a standalone element
- The LinkedList class contains one property that represents the head of the list
- A LinkedList object will initially have the head property as pointing to null (i.e. a new LinkedList will initially be empty)
- When we create Node objects to add to the list, we will insert through “head”
- The other functions in the LinkedList class provide capabilities to insert, update, search, and delete Nodes in the LinkedList

Linked Lists

Insert

- The algorithm for unordered linked list insert is as follows:
  - Create a new Node
  - Make the new Node's "next" property point to the "head" of the Linked List
  - Make the "head" of the Linked List the new Node

Linked Lists

```
public void insert(int value)
{
  Node newNode = new Node();
  newNode.data = value;
  newNode.next = head;
  head = newNode;
}
```

Linked Lists

Visualizing the insert

- The linked list will be in one of 2 situations: the list is empty or it is not
- If the list is empty, the head node will be pointing to null
- If the list is not empty, the head node will be pointing to a Node object
- This insert algorithm will insert the new node at the head of the list
- We'll examine this by first looking at inserting a new Node into an empty LinkedList

Linked Lists

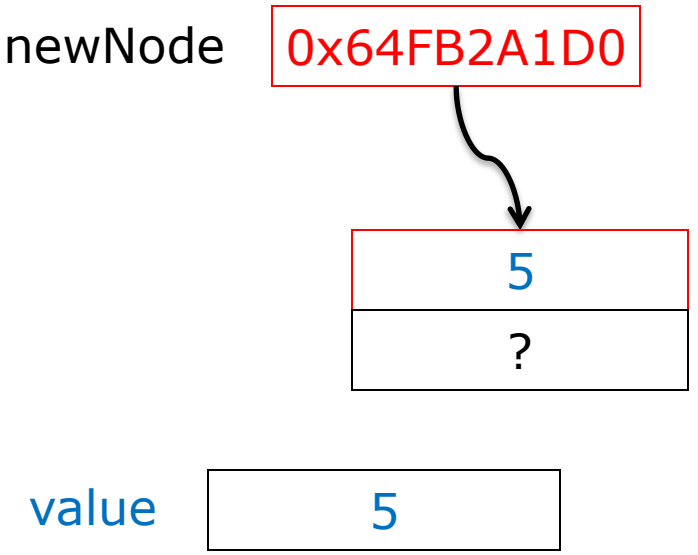
Create LinkedList object and insert a 5:

```
LinkedList list = new LinkedList();
list.insert (5);
```



The code in the LinkedList insert function executes as follows:

```
insert (int value)
{
   Node newNode = new Node();
   newNode.data = value;
   newNode.next = head;
   head = newNode;
}
```

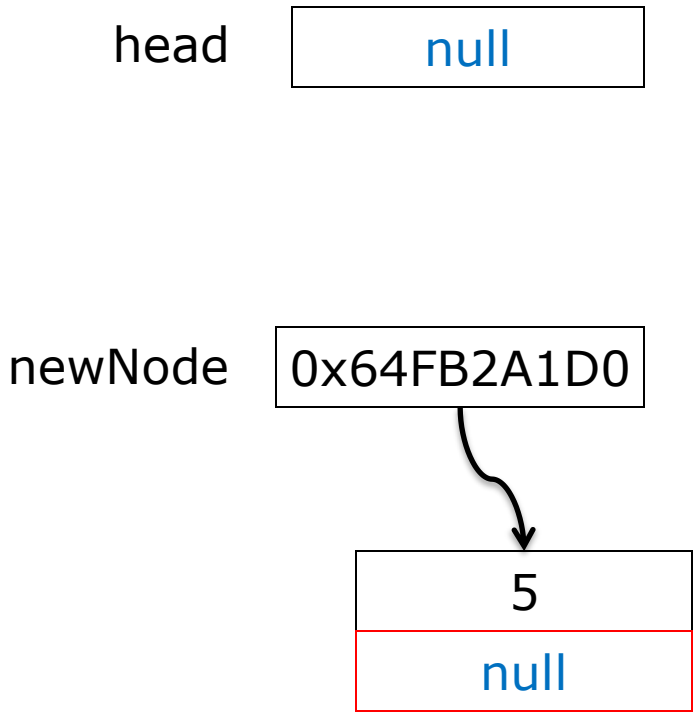


Linked Lists

Insert into blank list:

```
insert (5);

insert (int value)
{
   Node newNode = new Node();
   newNode.data = value;
   newNode.next = head;
   head = newNode;
}
```

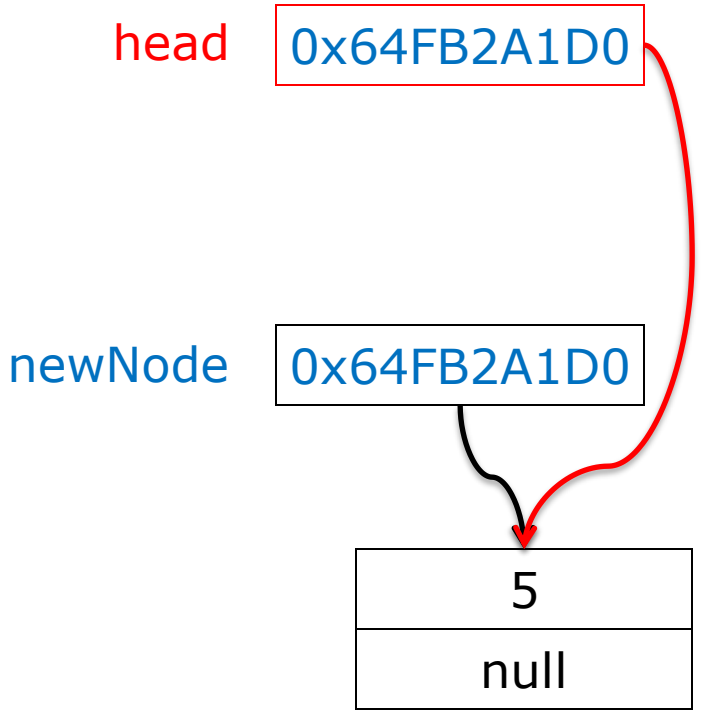


Linked Lists

Insert into blank list:

```
insert (5);

insert (int value)
{
   Node newNode = new Node();
   newNode.data = value;
   newNode.next = head;
   head = newNode;
}
```



Linked Lists

Visualizing the insert

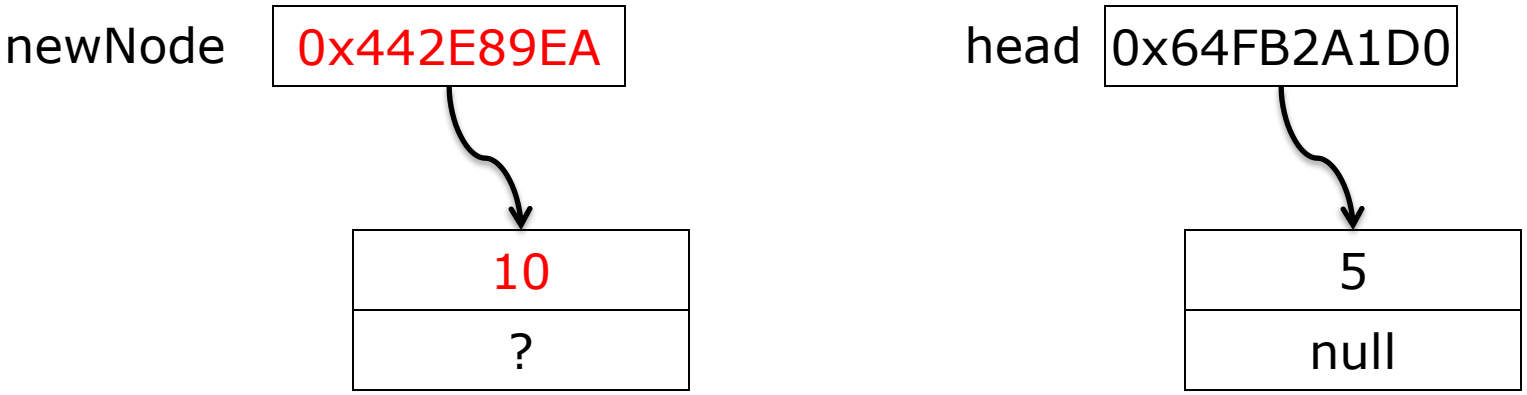
- At this point, the head property of the LinkedList now points to an existing Node
- Since we only added one Node, the “next” property of the head Node is pointing to null
- When we add another Node to the LinkedList, we will use the same insert function – the same sequence of steps will be performed
- What ends up happening is the new node object will become the new “head” of the list
  - We essentially add the new Node to the beginning of the LinkedList

Linked Lists

Insert into existing list:

```
list.insert (10);
```

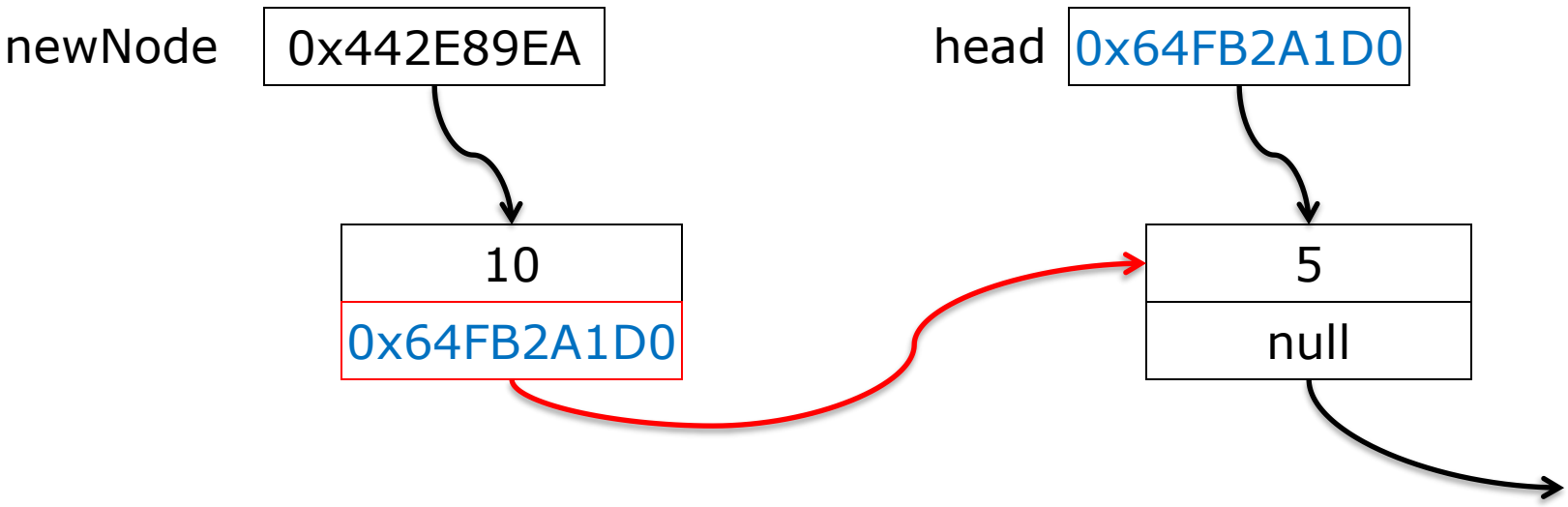
```
Node newNode = new Node();
newNode.data = value;
```



Linked Lists

Insert into existing list:

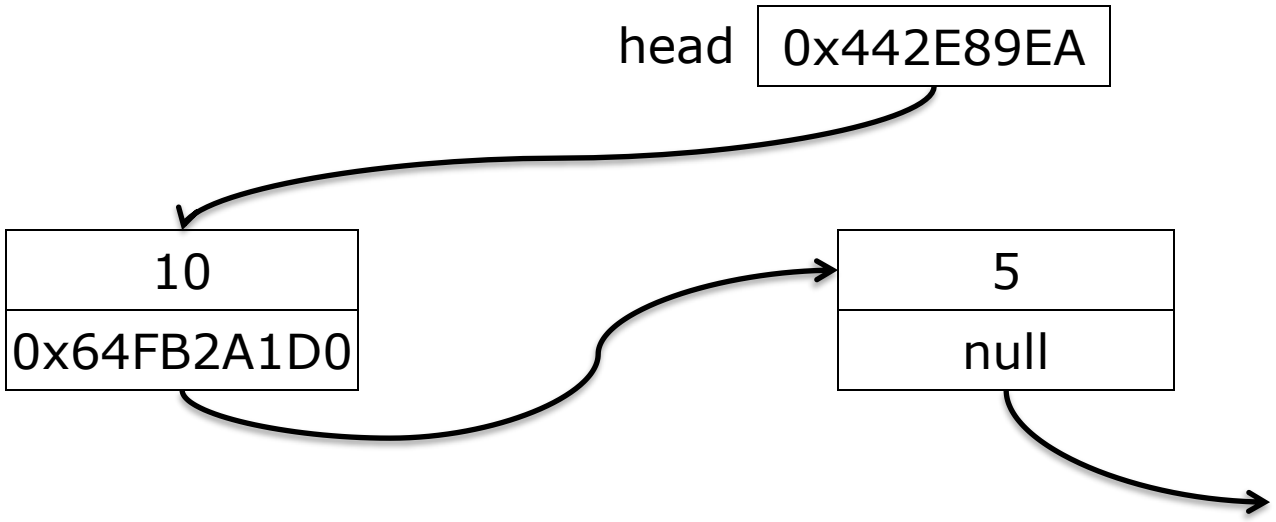
```
newNode.next = head;
```



Linked Lists

Insert into existing list:

```
head = newNode; // head is redirected to new head of
                // list
```




```
// Notice when the insert function ends, the
// "newNode" variable used in the function goes away
```

Linked Lists

Visualizing the insert

- With each insert performed, the “head” object takes on the new node being created
- The new node created has its next pointer reference the previous LinkedList through the “head” object
- This is the equivalent of doing an “insertFrontWithShift” in an array
- Notice here though, that we don’t do any shifting, we simply create a new Node, make it the new head and have its next pointer reference the previous head of the list

Linked Lists

Insert analysis

- Now since we used the compare operation in the insert analysis of the unsorted arrays, we have to do the same thing here
- As it turns out, there isn't even one compare operation performed
  - With the array, we needed to check if there is available space
  - Since memory is used only when needed, that check is not necessary (assuming there is enough memory as a whole)
- We stay at O(1) and we also only use memory when needed. The dynamic memory usage is a win over arrays with Linked Lists so we can assume if performance is equal, the main benefit is the dynamic memory usage
- Notice with this form of insert, the data in the list is unordered - this means when looking at search, we are searching through an unordered set of data

Linked Lists

Search

- Unordered arrays search at O(n), and you can probably guess with Linked Lists, it's going to be the same because of the nature of the data
- The algorithm for unordered linked list search:
  - Create a temporary Node that points to the head of the list
  - While the temp Node is not null
    - If the data at that node is the value you are searching for, return the temp Node – the value was found!
    - Else, go to the next Node by assigning temp to the node's "next" property – this process is called "traversing" the list
  - If we reach this point, the value was not found in the list and we return null

Linked Lists

Unordered search:

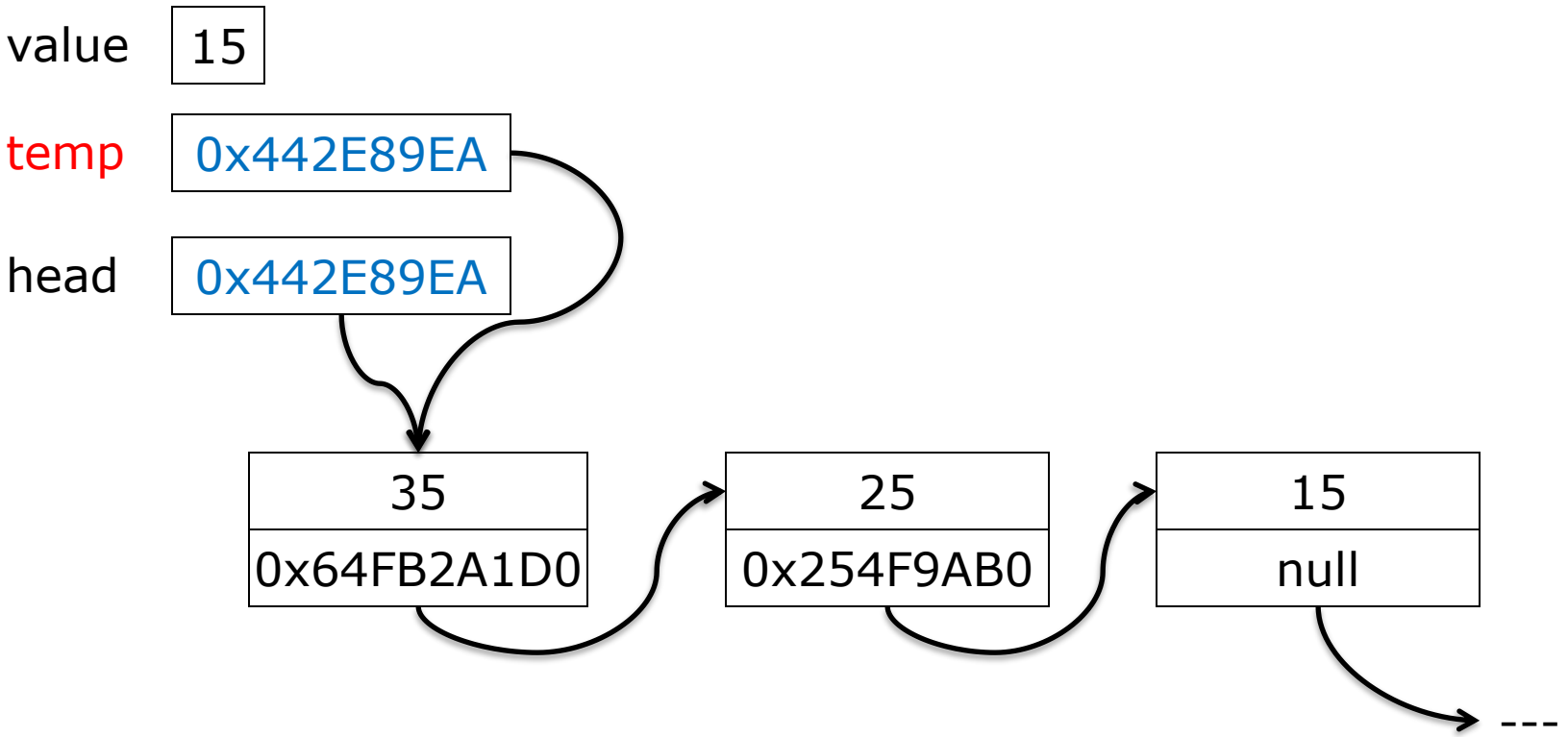
```
public Node search(int value)
{
   Node temp = head;

   while (temp != null)
   {
       if (temp.data == value)
              return temp;
       temp = temp.next;
   }

   return null;
}
```

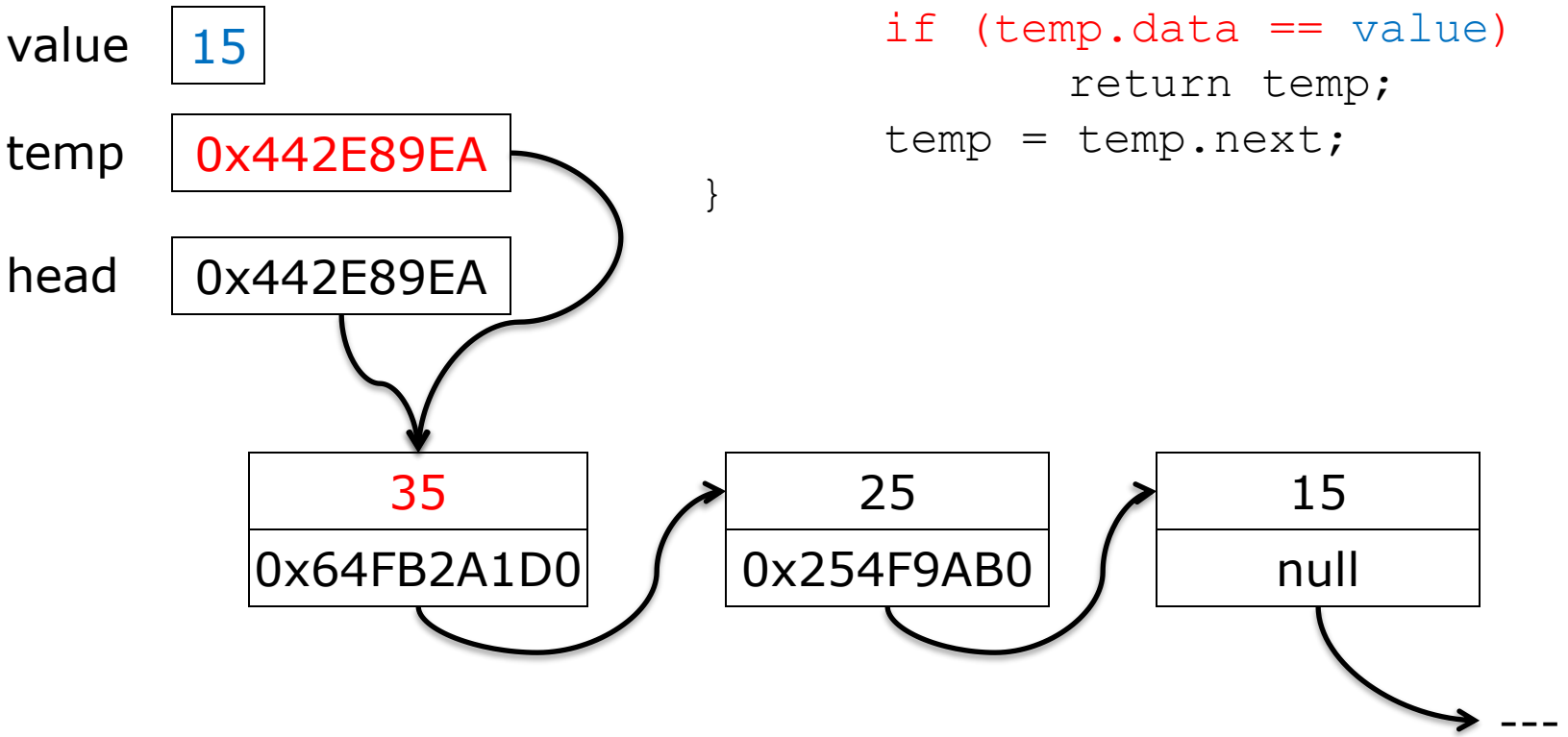
Linked Lists

```
Node temp = head;
```



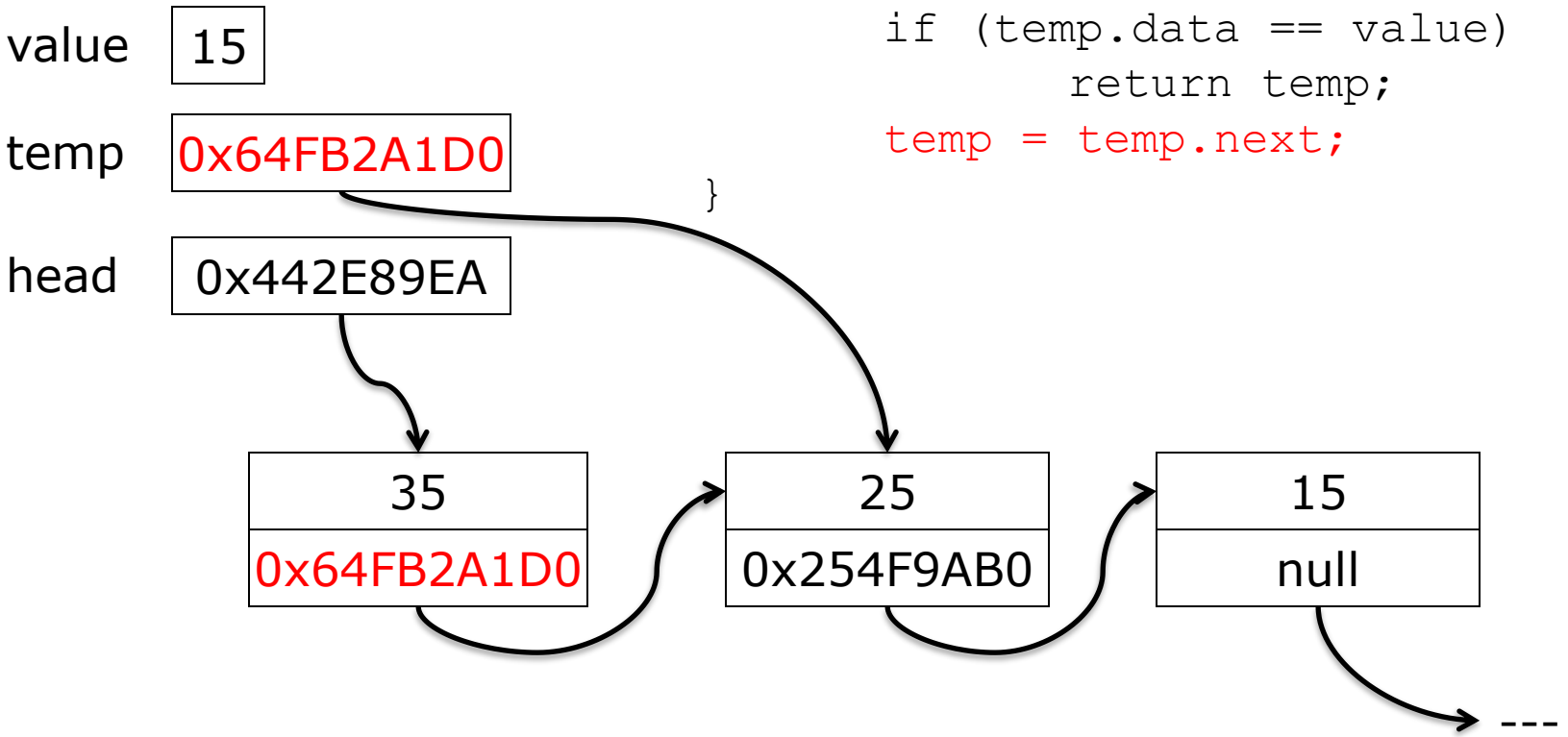
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



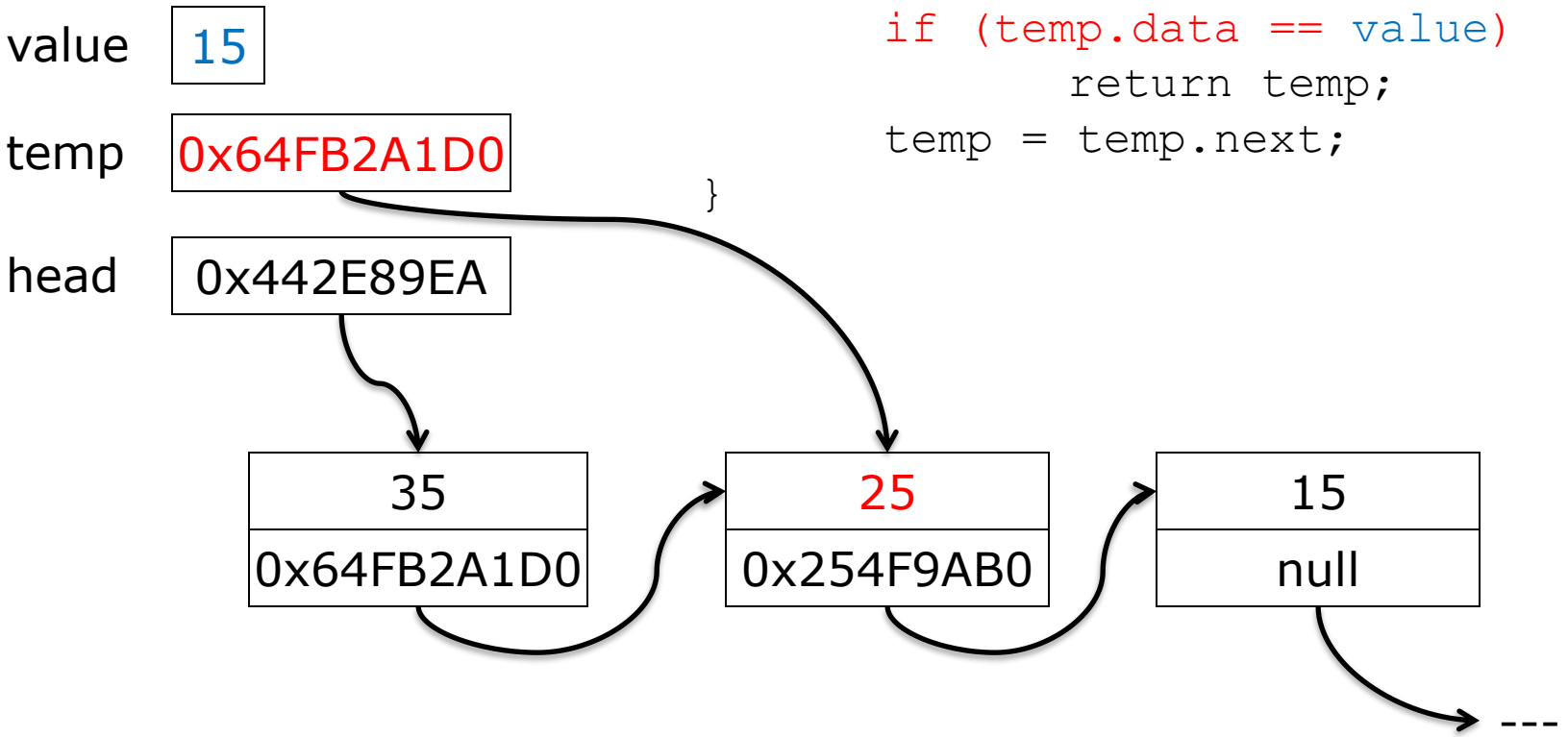
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



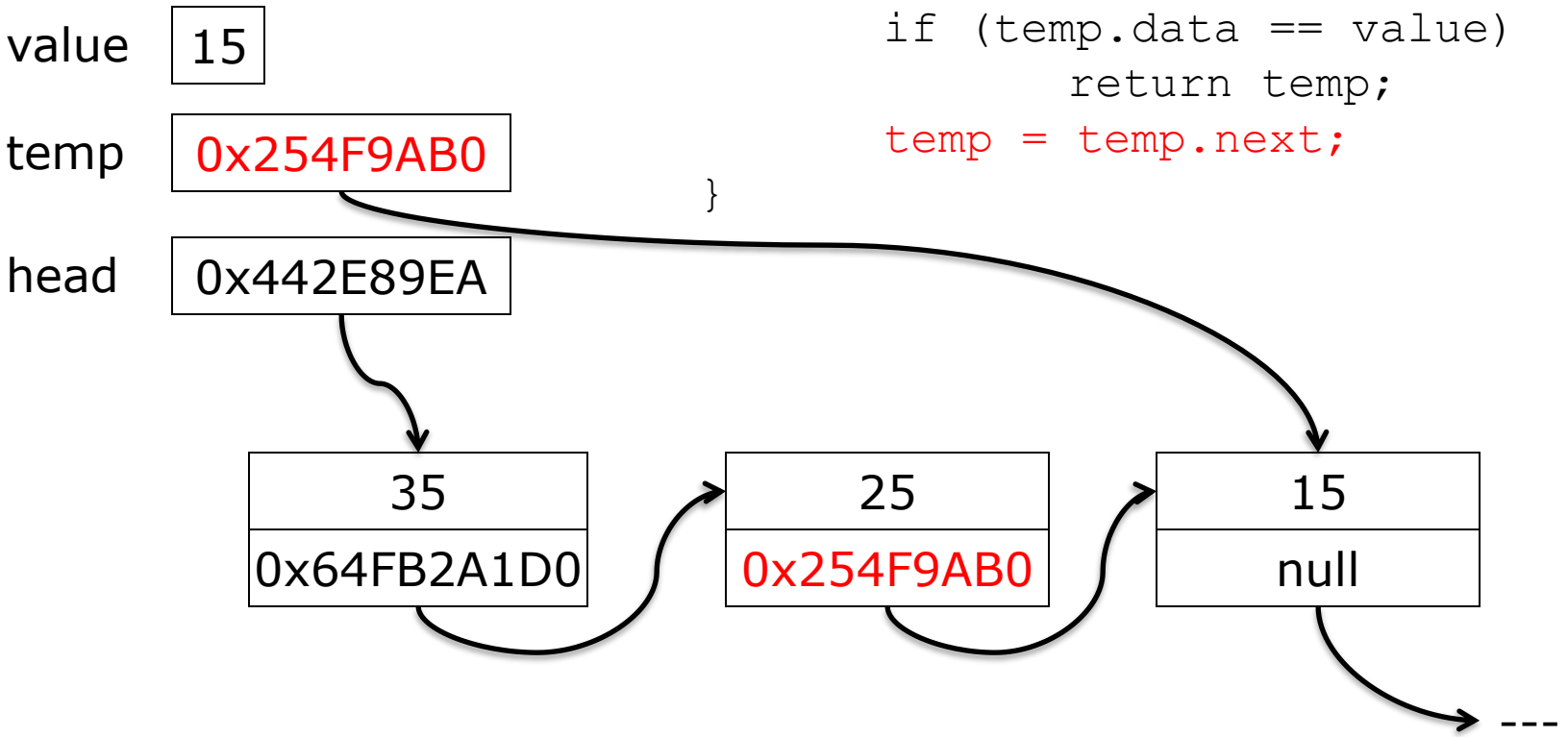
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



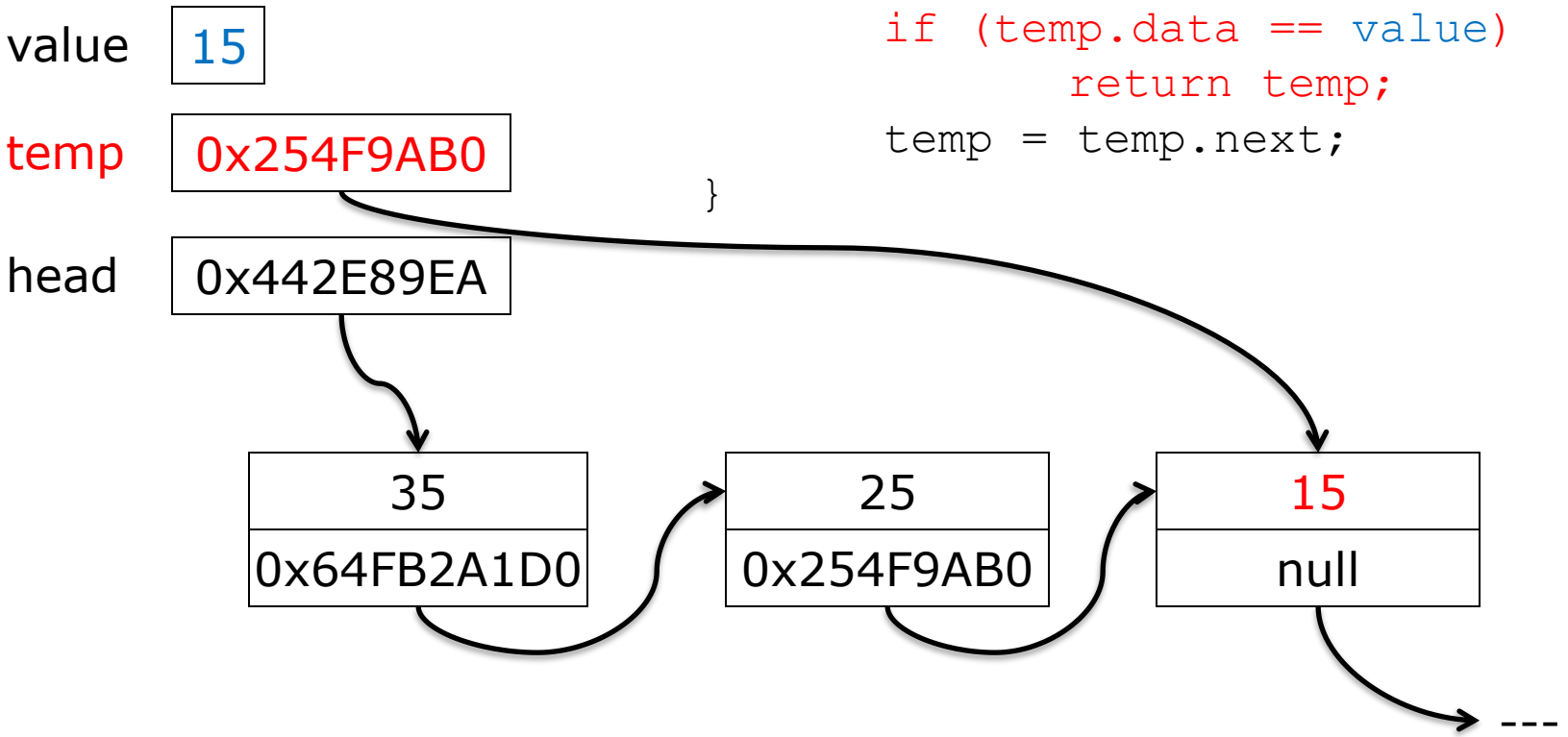
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



Linked Lists

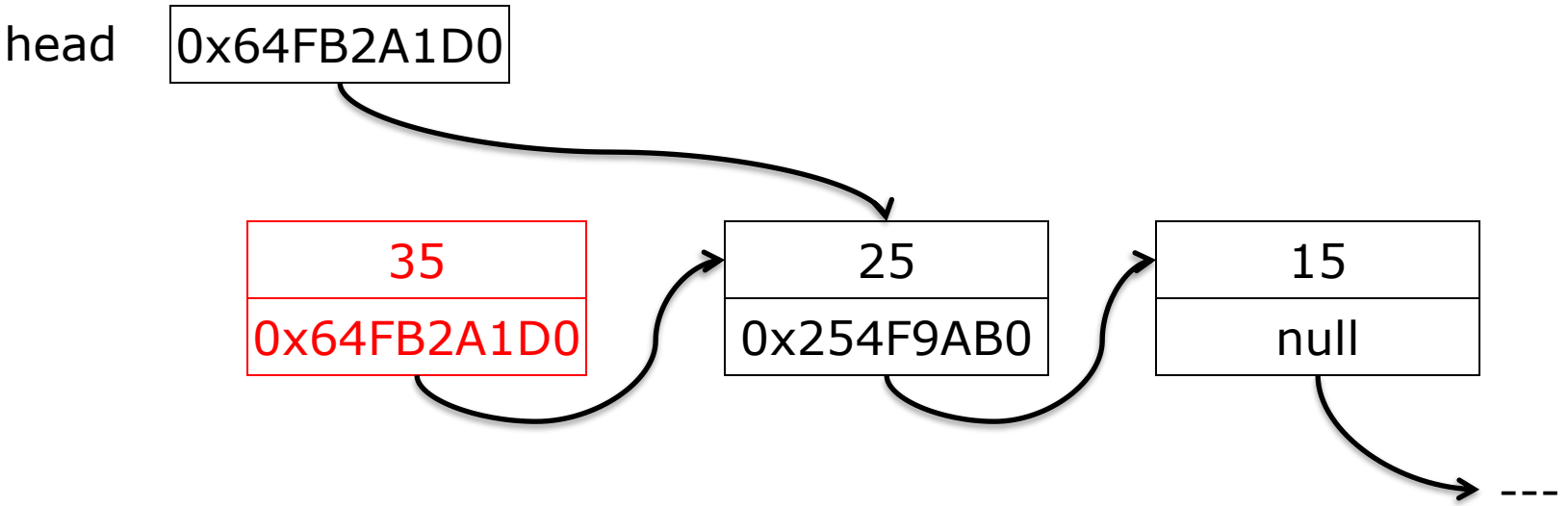
Visualizing the search

- The temp pointer is used to traverse the LinkedList, checking each element's data value to see if we found the Node
- If we find the Node with the search value, we stop the function there and return the Node
- If we don't find the Node with the search value, temp will end up with a value of null and the while loop will end
- The key with the search algorithm is the use of temp itself
- When the search function executes, the head pointer never changes – if it did, we would lose visibility to the head of the list

Linked Lists

Search analysis

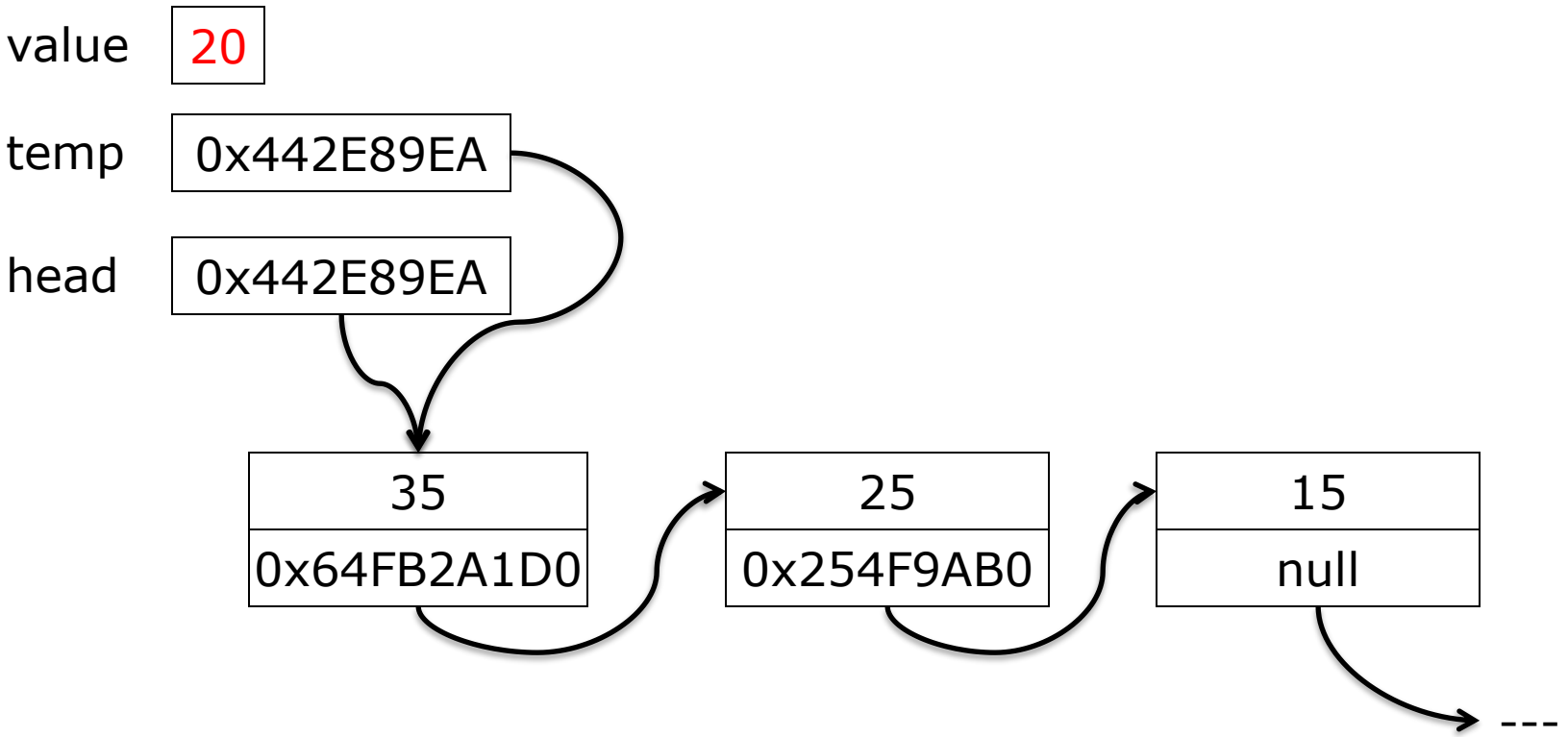
- What would happen if we traversed the list using the head variable with "head = head.next" versus "temp = temp.next"



- Node 35 would have nothing referencing it and we would end up losing that node in memory
- Let's next look at an example of searching a list and not finding a Node with the search value

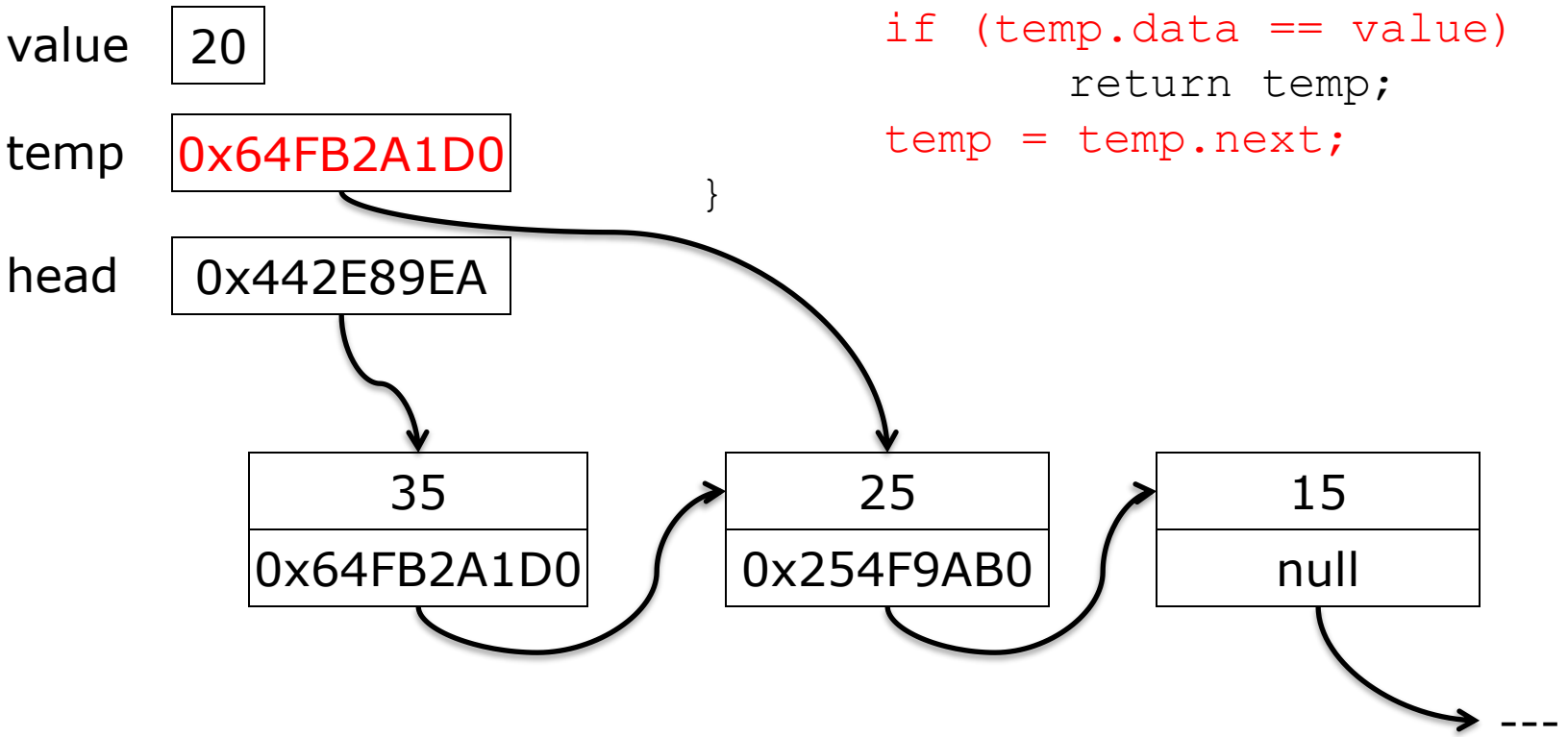
Linked Lists

```
Node temp = head;
```



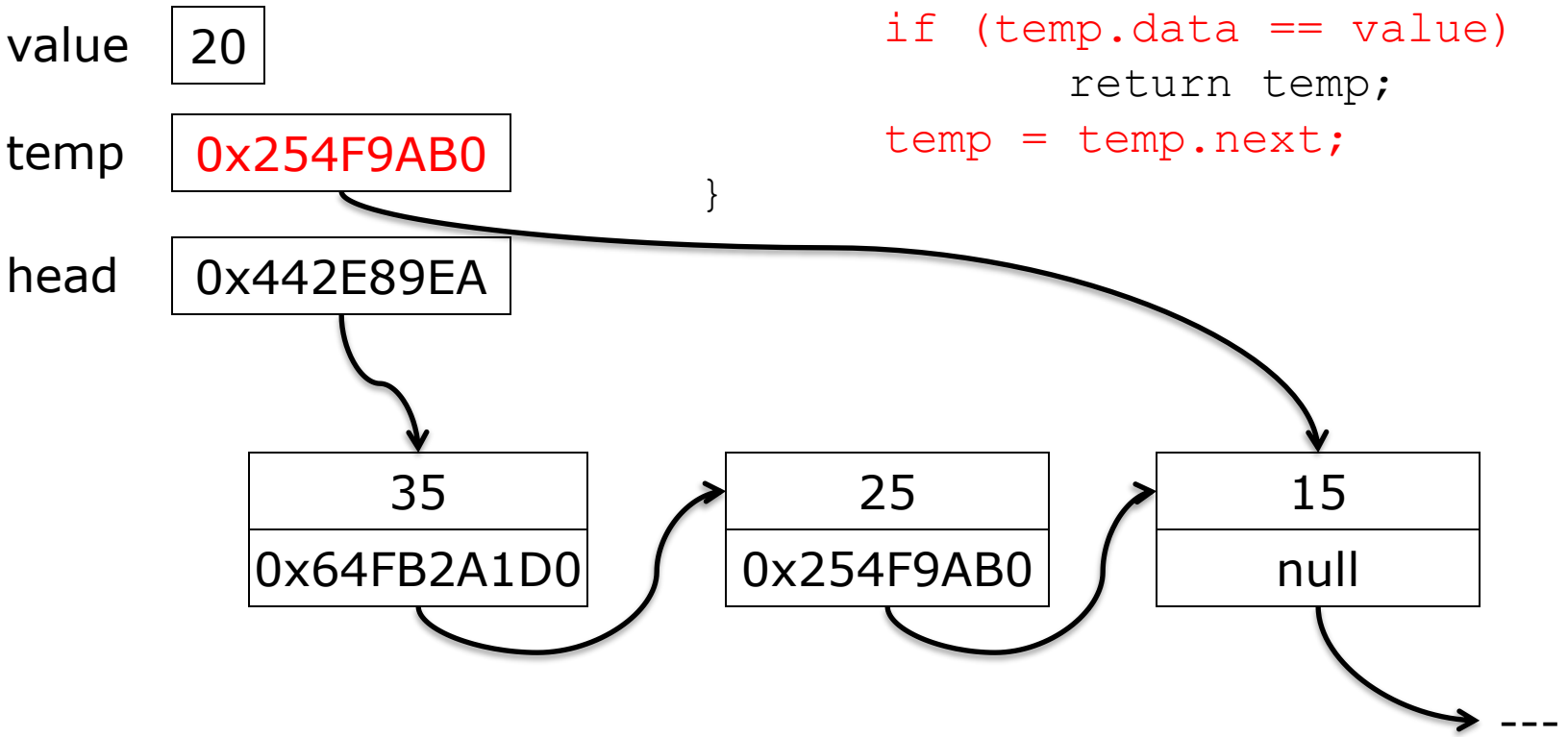
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



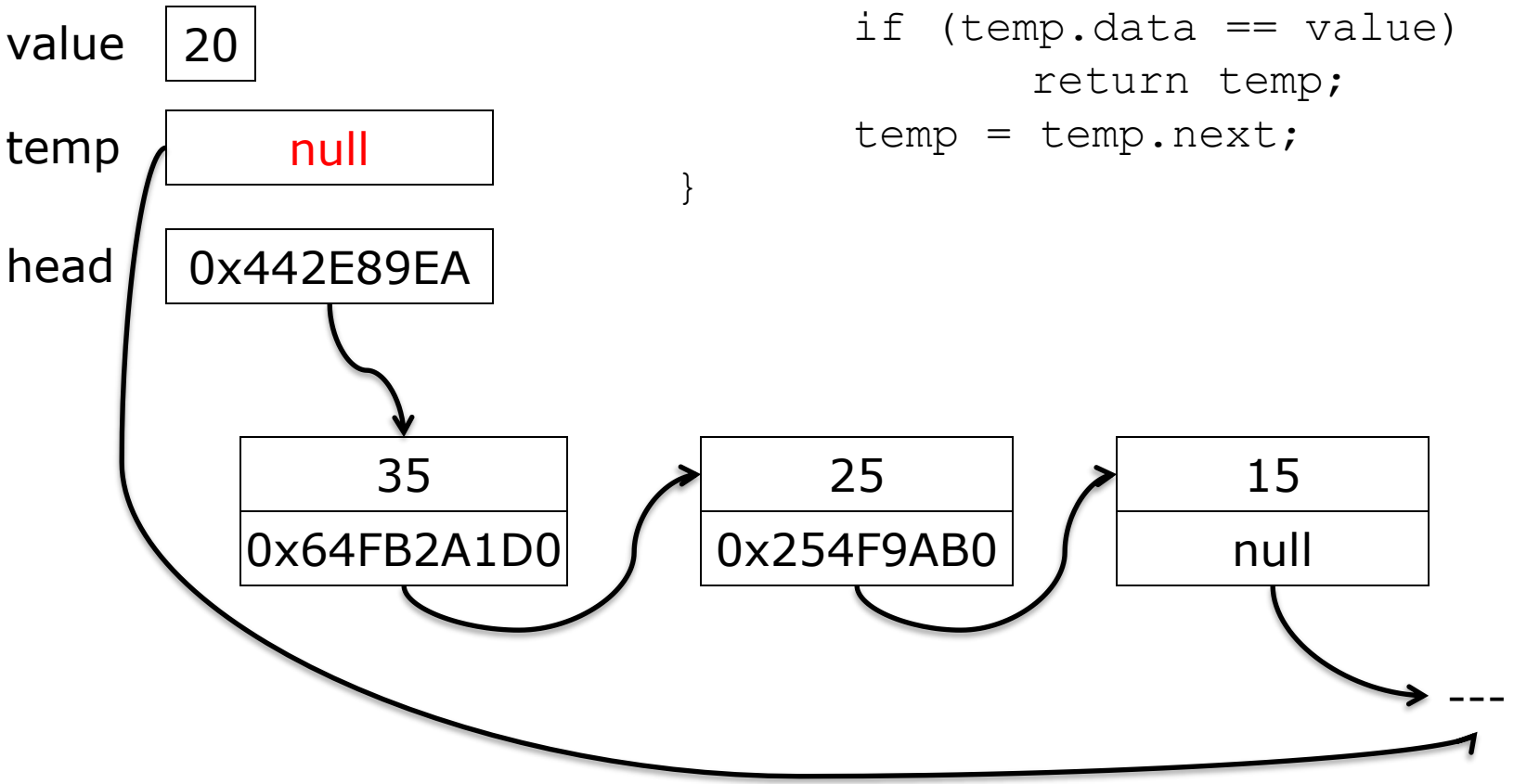
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



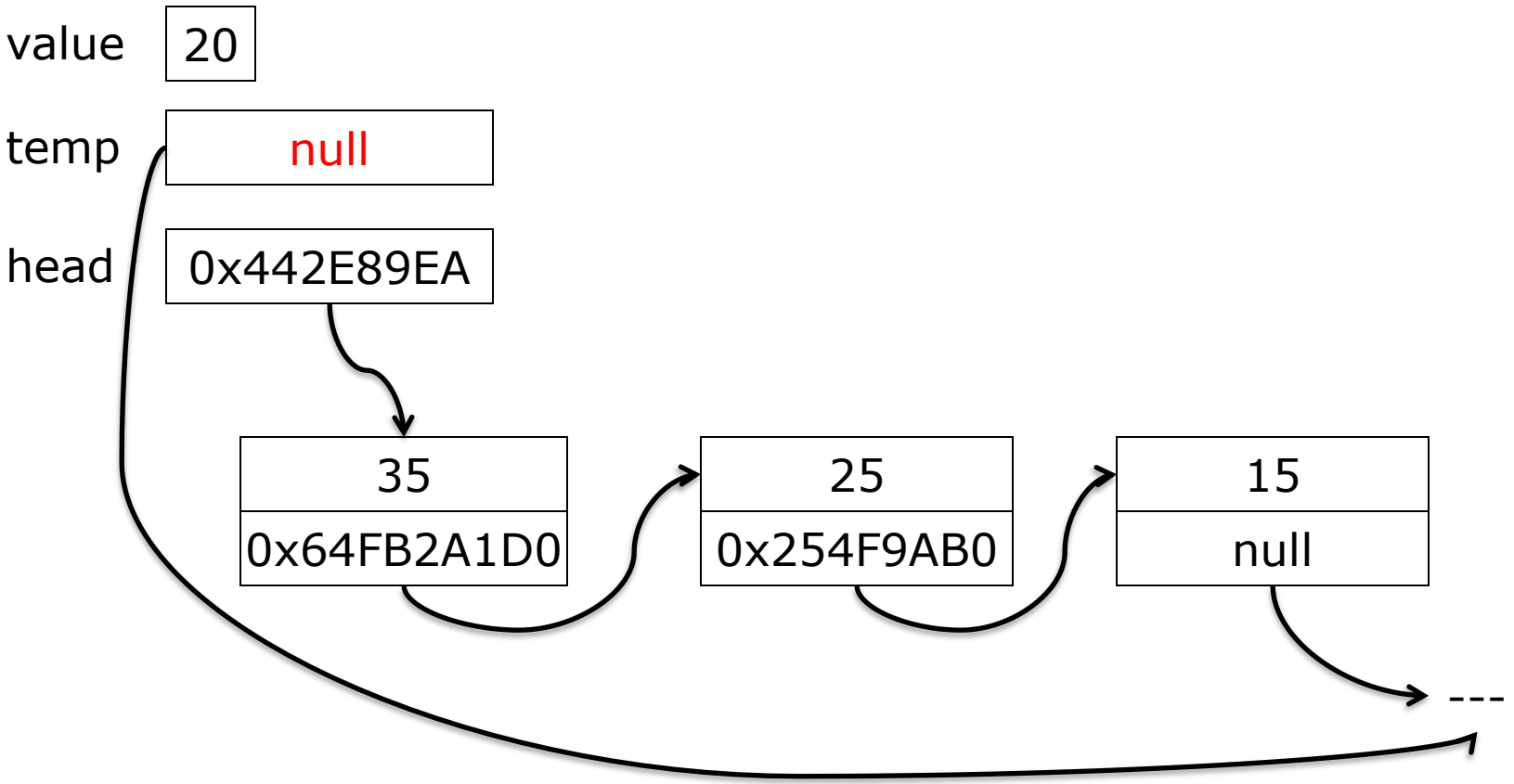
Linked Lists

```
while (temp != null)
{
        if (temp.data == value)
                return temp;
        temp = temp.next;
}
```



Linked Lists

```
return null;
```



Linked Lists

Search analysis

- We have a loop using a comparison operation and in the worst case (if the search value does not exist), the performance is O(n) so search is the same as an unordered array
- Update and delete both use a search so we're also at least looking at an O(n) performance here as well

Linked Lists

What about the append function?

- In an unordered array, the append is an O(1) operation
- For a Linked List, append requires finding the end of the list
- For a list with only a head node pointer, this will require “traversing” the list like a search to get to the end and attaching the new node
- The search in this case is to find the node in the list where the “next” node is null (i.e. the last node in the list)

Linked Lists

```
public void append(int value)
{
   Node newNode = new Node();
   newNode.data = value;
   newNode.next = null;

   if (head == null)
        head = newNode;
   else
   {
        Node temp = head;
        while (temp.next != null)
               temp = temp.next;
        temp.next = newNode;
   }
}
```

Linked Lists

Analysis

- Because the append traversal is like a search, it will perform at O(n)
- Question: is it possible to improve the append performance so that it is O(1)?
  - Answer is in Queues implemented as a Linked List…

- Let's next look at the update function
- Update will use the search function and if it finds the value to change, the return value will be a reference to that Node and we can simply change its data property to the new value
- If the search does not find the value to change, the return value will be null – no update can then be performed

Linked Lists

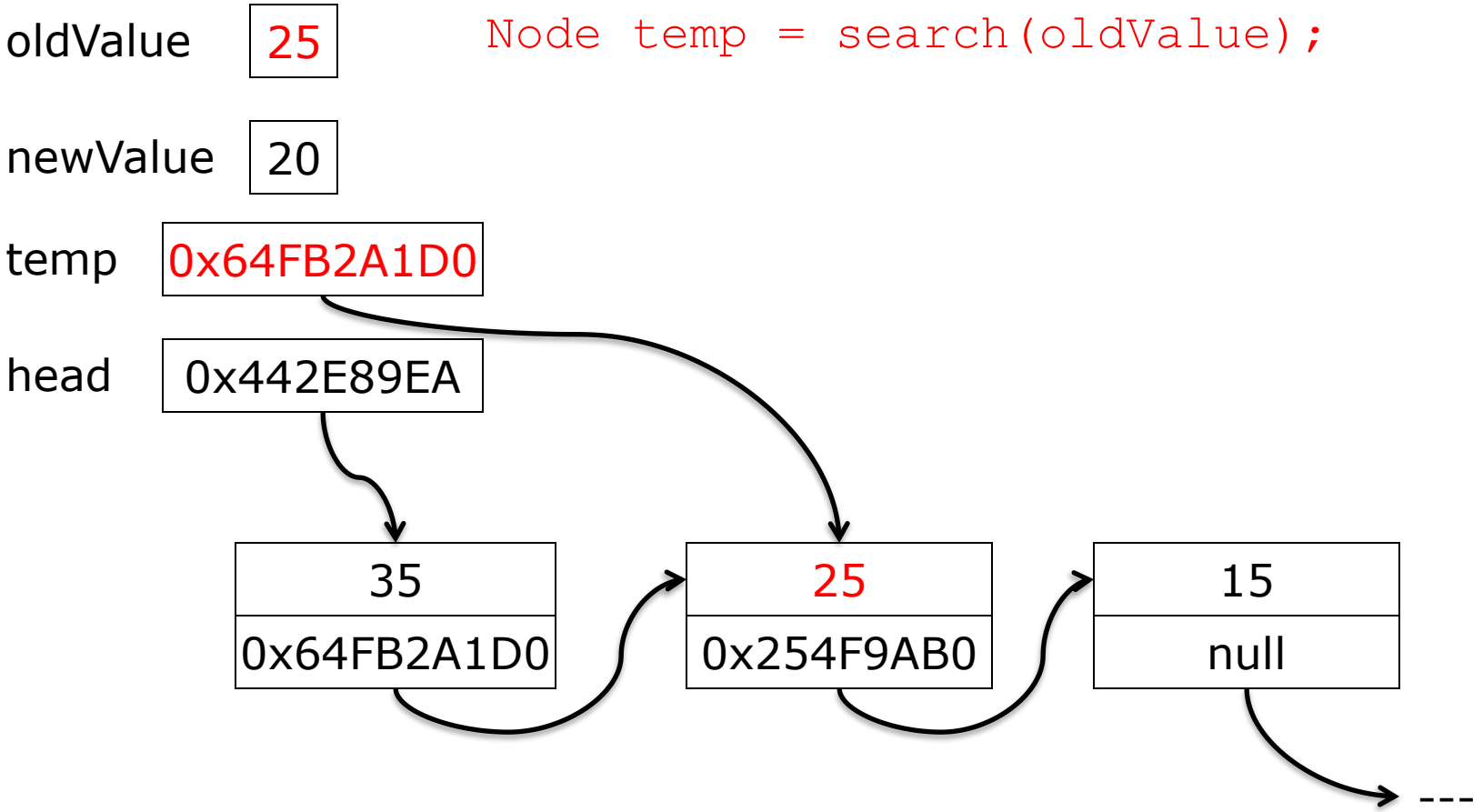
Unordered update:

```
public boolean update(int oldValue, int newValue)
{
   Node temp = search(oldValue);

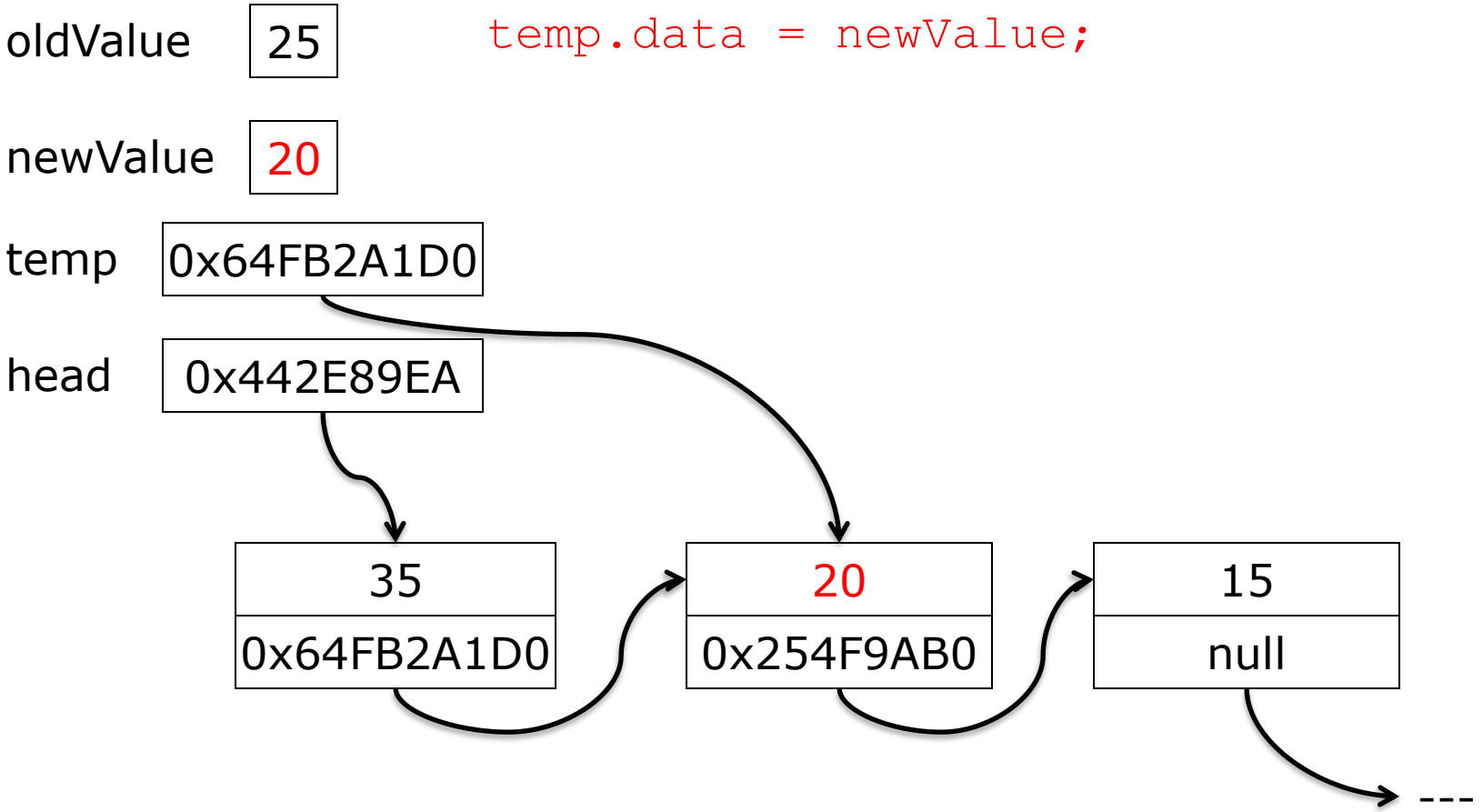
   if (temp == null)
       return false;

   temp.data = newValue;
   return true;
}
```

Linked Lists



Linked Lists



Linked Lists

Update analysis

- We have another O(n) algorithm. This makes 3 out of the 4 algorithms the same performance between unordered arrays and linked lists. It's starting to look like Linked Lists are better suited for unordered data structures because of the dynamic memory usage (if memory management is of primary concern)
- Let's look at delete. A pattern you will see with all data structures we look at is that the delete function tends to be the most complex, and that is true here
- With the linked list, there are 4 situations to consider when deleting a node from the list

Linked Lists

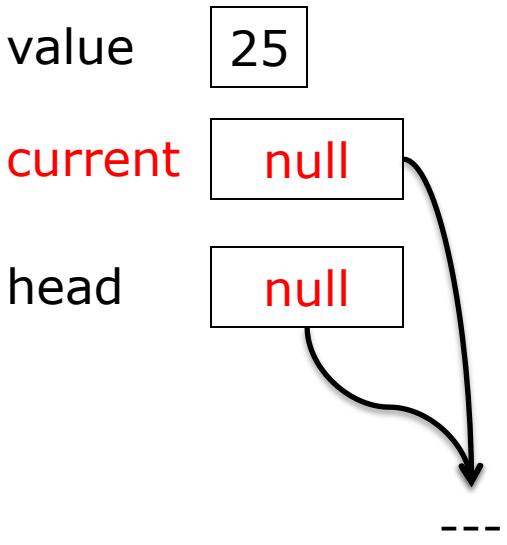
Delete analysis, situation 1

- With delete functions for the data structures, you usually see them handled by dealing with 4 key situations:
  1. An empty list
  2. A list where the element to delete is the first
  3. A list where element to delete is not in the list
  4. A list where the element to delete is elsewhere in the list

- In the first situation, this is an easy one to handle. The list is empty, so we're done! However, it is very important to address this check first:
  - An empty list means the head is null
  - Whenever you access an object's property like "current.data", the code will first dereference current
  - If current is null, it can't go anywhere making the ".data" part of the code unusable and results in an "accessing null pointer exception"
  - The code syntax, however, is correct making it a run time error
  - This is easy to miss during debugging because it is natural to test the code with lists that contain data

- Never forget to check if a list is empty first when working with dynamic memory data structures like LinkedLists!

Linked Lists




```
public boolean delete(int value)
{
        Node current = head;

        // Situation #1
        if (current == null)
                return false;
```

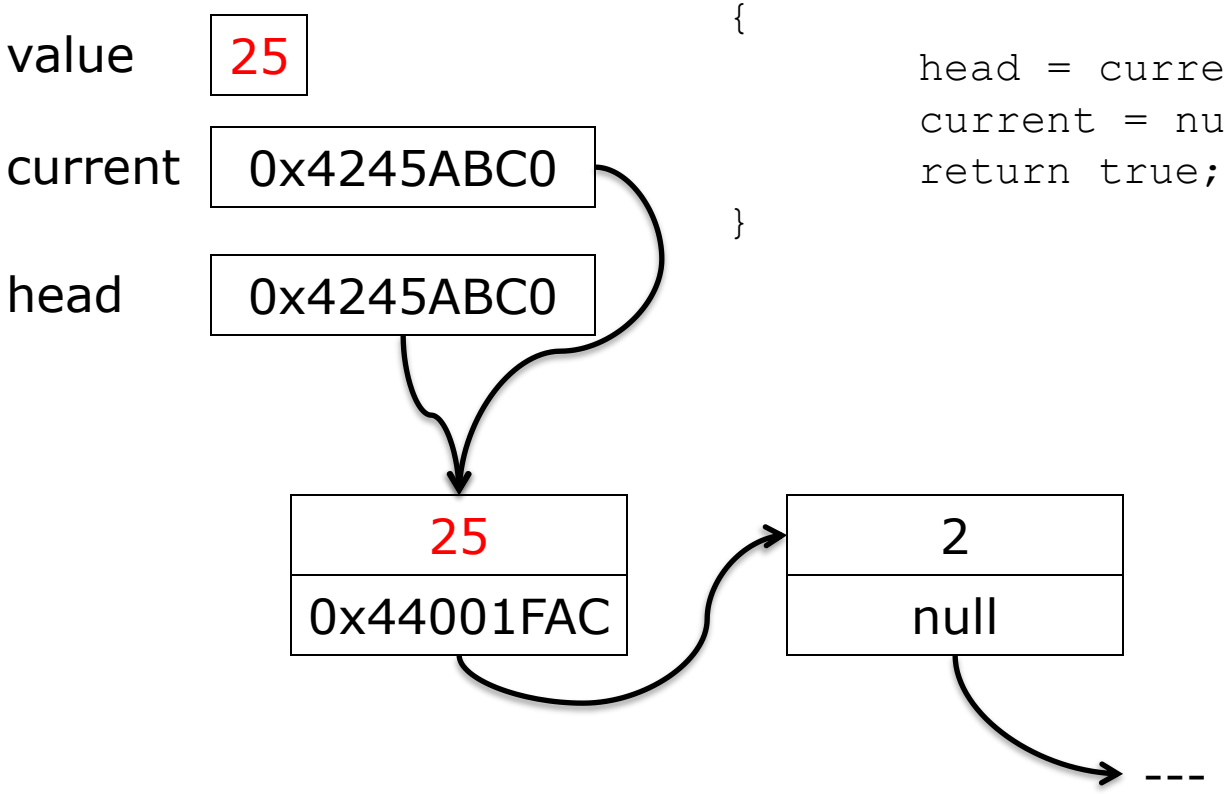
Linked Lists

Delete analysis, situation 2

- The other situations can now safely assume at least one element exists in the list
- Situation 2 is where the element to delete is the at the head
- A proper delete here means we have to reassign the head pointer to a new Node
- For this to happen, the new head Node is the current head's "next" Node

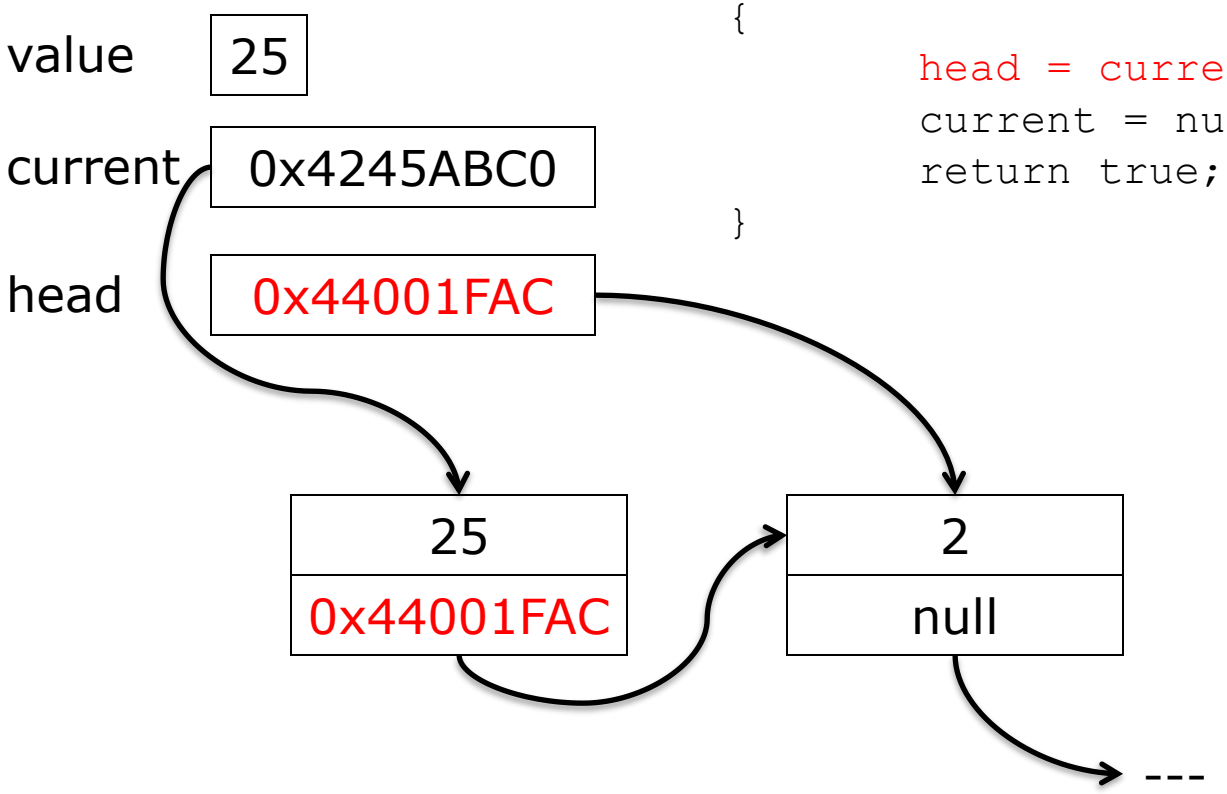
Linked Lists

```
// Situation #2
if (current.data == value)
{
        head = current.next;
        current = null;
        return true;
}
```



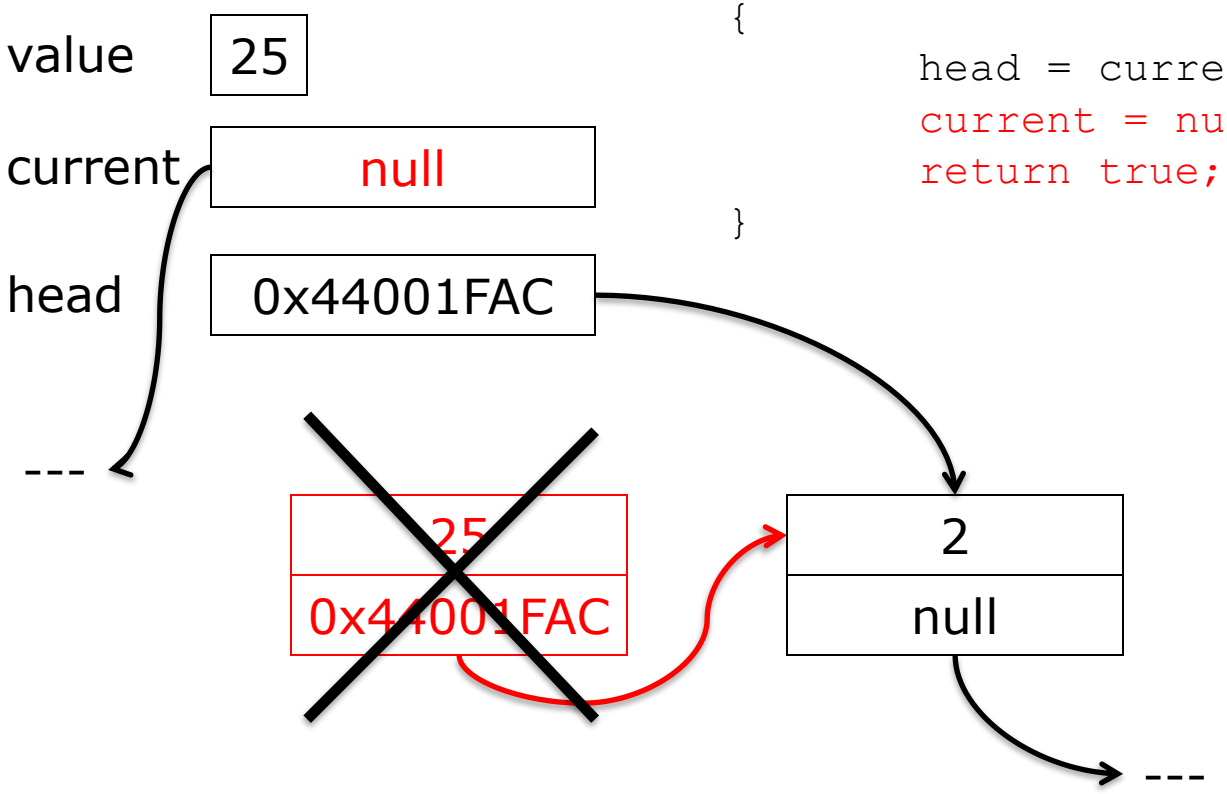
Linked Lists

```
// Situation #2
if (current.data == value)
{
        head = current.next;
        current = null;
        return true;
}
```



Linked Lists

```
// Situation #2
if (current.data == value)
{
        head = current.next;
        current = null;
        return true;
}
```



Linked Lists

Delete analysis, situation 2

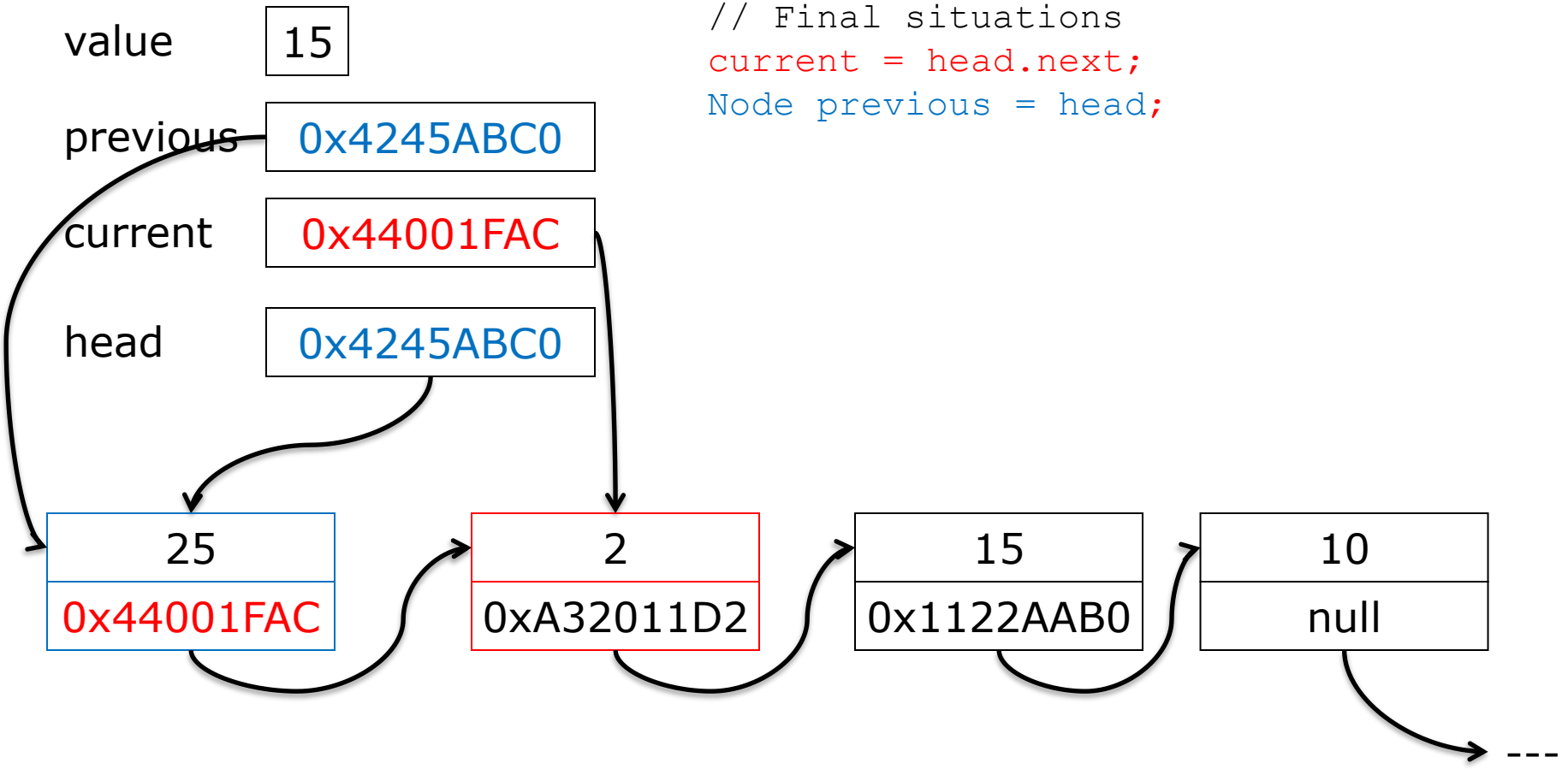
- Note that when we say “current = null;” this effectively deletes that Node. In Java, objects with no pointers referencing it will automatically be deleted
- in C++ though, you have to explicitly free the memory space with “delete current;”
- This again stresses the importance of drawing pictures when it comes to the joys of pointers and dynamic memory management
- Setting objects to null incorrectly can lead to inconsistent and unexpected results during testing

Linked Lists

Delete analysis, situations 3 and 4

- In the remaining situations, we have to traverse the list to see if the element exists (which is O(n))
- If we traverse the list and don't find the element, we'll reach null and we're done (situation 3)
- Finding the element to delete, though, requires some additional pointer management
- As we traverse the list, we cannot just set the "current" Node to the next one though because we need visibility to the Node behind the current one in order to properly delete the Node and maintain the other Nodes in the list
- We do this by maintaining a "previous" pointer along with "current" and start by setting "previous" equal to the head node and "current" equal to head.next
- During the loop, if the current Node is not the one to delete, "current" and "previous" pointers traverse to their respective next nodes

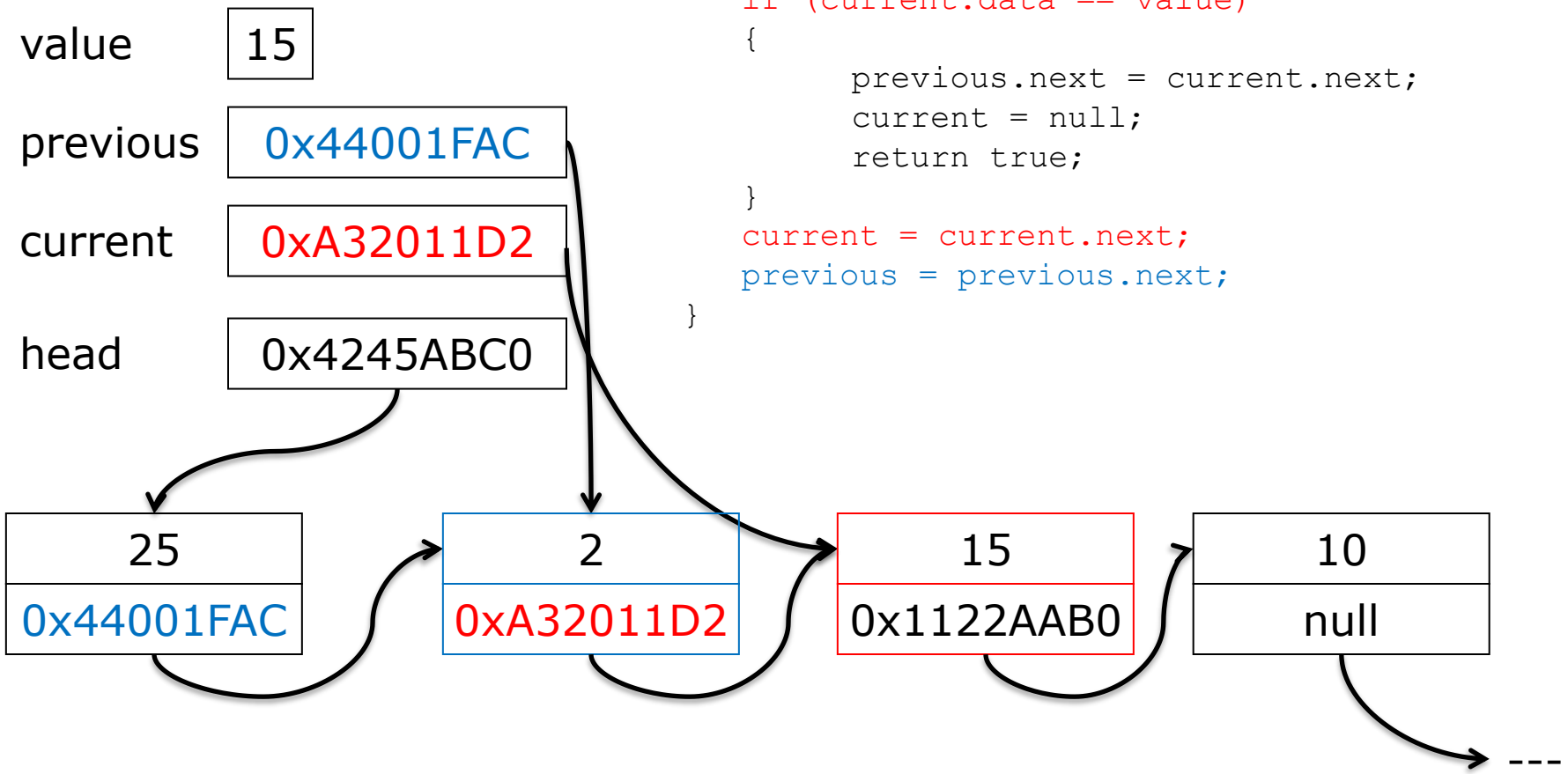
Linked Lists




```
// Final situations
current = head.next;
Node previous = head;
```

Linked Lists

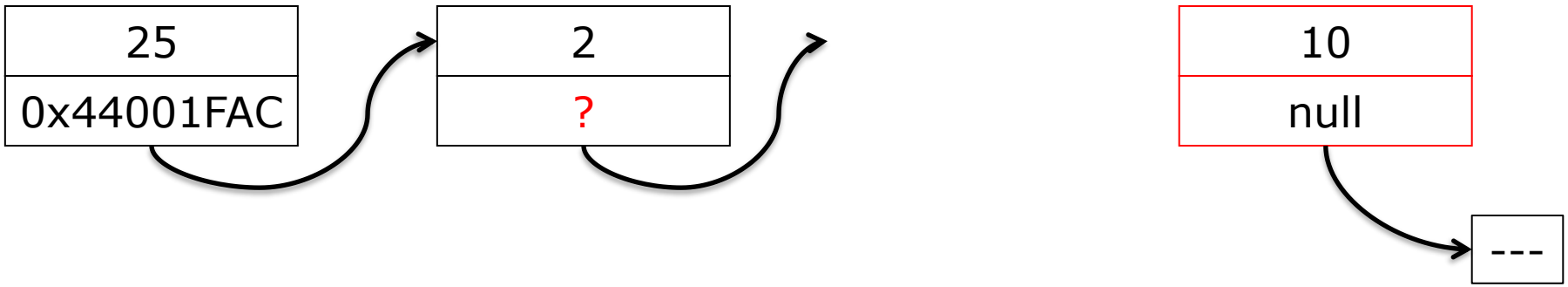
```
while (current != null)
{
    if (current.data == value)
    {
        previous.next = current.next;
        current = null;
        return true;
    }
    current = current.next;
    previous = previous.next;
}
```



Linked Lists

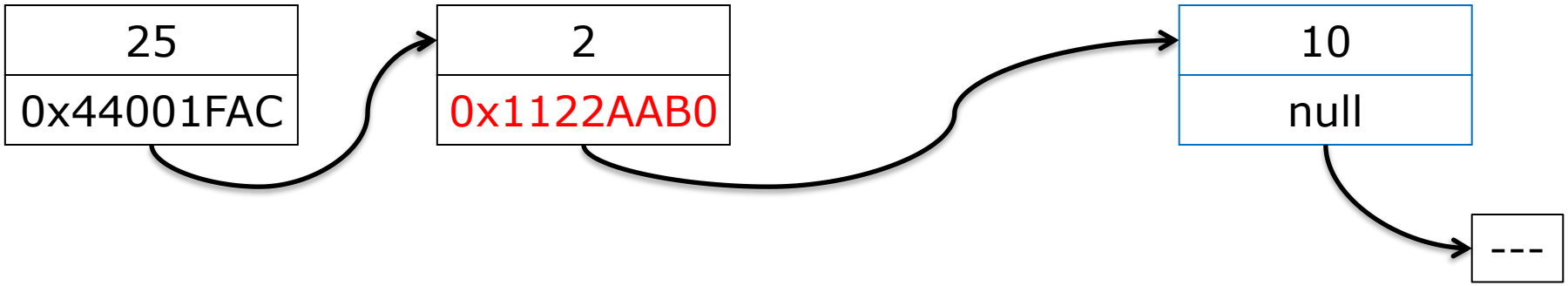
Delete analysis

- When the target node is found, the current node cannot just be set to null as the LinkedList would now be broken
  - Node 2's next pointer references nothing
  - Node 10 has nothing referencing it and is now inaccessible



Linked Lists

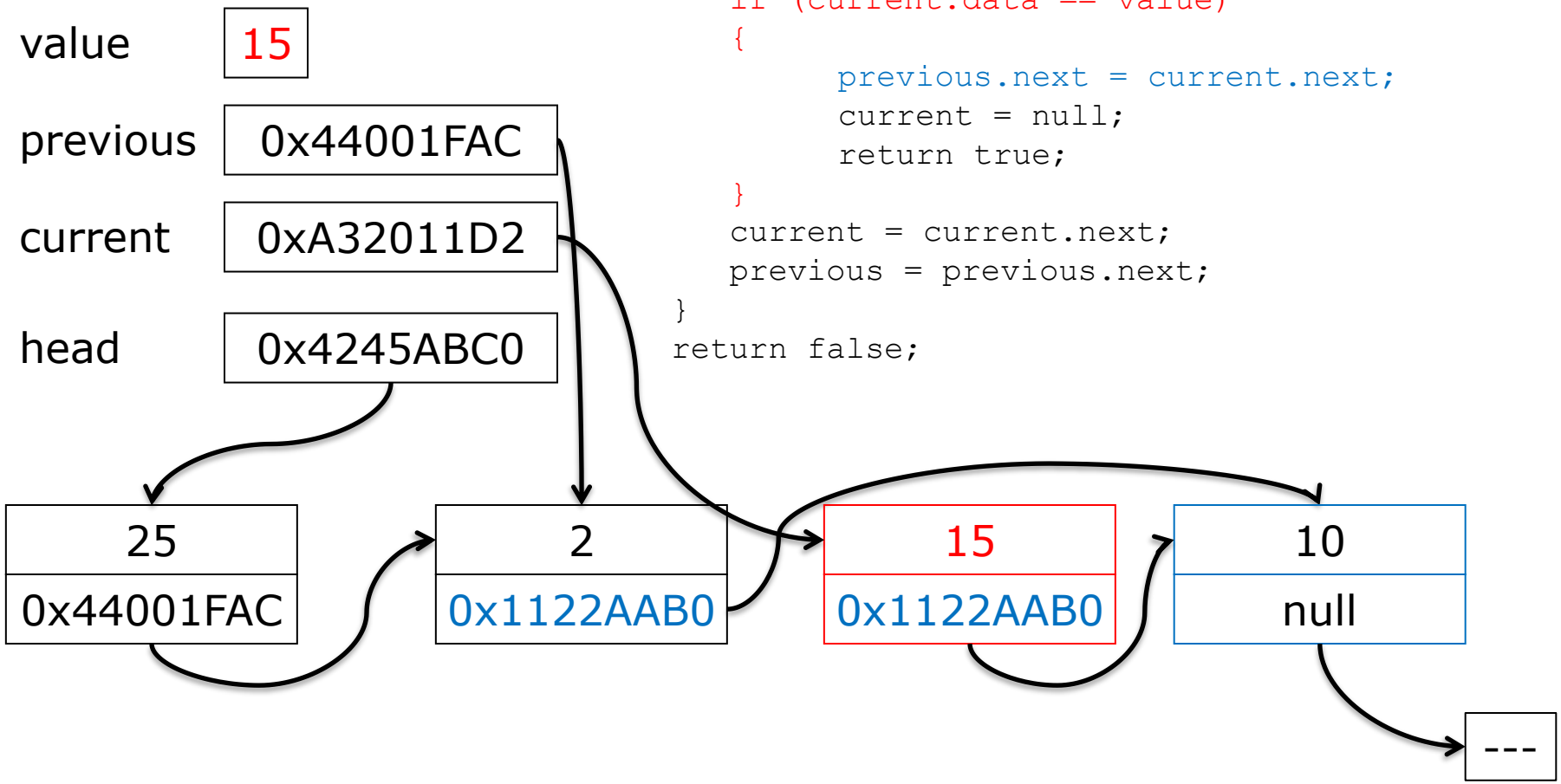
- To properly delete the node, the neighboring nodes must be connected to each other
- The "previous" Node's next pointer must first be set to the "current" Node's next pointer
- The end result should look like this



- Let's see how the code does this line by line when it finds the node to delete

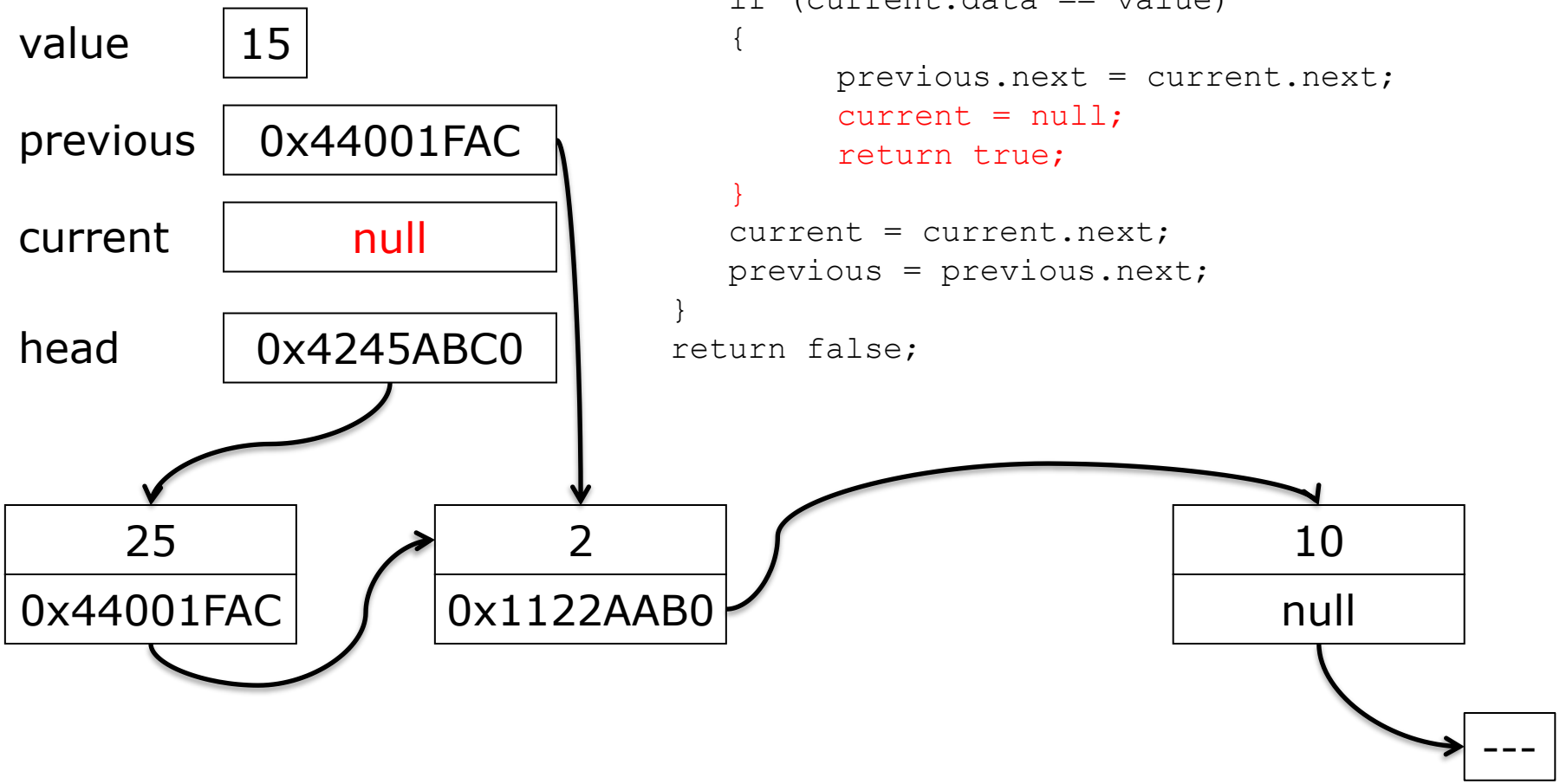
Linked Lists

```
while (current != null)
{
    if (current.data == value)
    {
        previous.next = current.next;
        current = null;
        return true;
    }
    current = current.next;
    previous = previous.next;
}
return false;
```



Linked Lists

```
while (current != null)
{
    if (current.data == value)
    {
        previous.next = current.next;
        current = null;
        return true;
    }
    current = current.next;
    previous = previous.next;
}
return false;
```



Linked Lists

Delete performance

- As far as Big O categories go, this is another O(n) performance (same as worst case search)
- Putting minute details aside between these 4 algorithms for unsorted arrays and Linked Lists, the performances are the same
- Linked Lists give you dynamic memory allocation so in situations where that is important, use of them is preferred over arrays
- The complete delete function code is detailed on the next 2 slides

Linked Lists

```
public boolean delete(int value)
{
   Node current = head;

   // Situation #1
   if (current == null)
       return false;

   // Situation #2
   if (current.data == value)
   {
       head = current.next;
       current = null;
       return true;
   }
```

Linked Lists

```
    // Final situations
    current = head.next;
    Node previous = head;
    while (current != null)
    {
        if (current.data == value)
        {
            previous.next = current.next;
            current = null;
            return true;
        }
        current = current.next;
        previous = previous.next;
    }

    return false;
}
```

Unordered Arrays vs. Linked Lists

Summary:

Worst Case	Unsorted Arrays	Unsorted Linked Lists
Search	O(n)	O(n)
Insert	O(1)	O(1)
Update	O(n)	O(n)
Delete	O(n)	O(n)
	Static memory usage	Dynamic memory usage

Linked Lists

Sorted linked lists theory

- The O(n) in the worst case of the sorted array functions is due to a need to shift all elements in the worst case
- With a linked list, we have already seen with the insert and delete algorithm that with proper node management, there is no “shift” eliminating the O(n) performance
- The initial thought, then, is that the performance for sorted linked lists is reduced to O(log n) for all 4 functions because the binary search is all that is required and there is no shifting of elements
- Question: Is this a correct assessment?

Linked Lists

Unfortunately, no

- Because of the nature of the Linked List structure for a Node only having visibility to its next Node, determining and directly accessing the “middle” of the chain of nodes is challenging
- Modifying the Linked List structure may help, but then you will have to find a way to map the entire list to simulate direct random access. At that point, you basically have an array
- The key difference between sorted arrays and sorted linked lists is that binary search can only be applied with arrays which leads to O(log n) performance
- Because binary search cannot be performed on linked lists, even if the data is sorted, its performance is still O(n)

Sorted Arrays vs. Linked Lists

Summary:

	Worst Case (Arrays)	Worst Case (LL)
Search	O(log n)	O(n)
Insert	O(n)	O(n)
Update	O(n)	O(n)
Delete	O(n)	O(n)

Linked Lists

Then what good are they?

- If the order of the data is not necessary, linked lists and arrays perform the same, but linked lists allow for dynamic memory usage
- If order matters, arrays are better especially if search is involved (so far…)
- Order in the list can also take different forms. Perhaps there are other ways to order data and still take advantage of the dynamic memory usage that linked lists use – that will be our next topic with Stacks and Queues