



CS2201

# Program Design and Data Structures

Fall 2025

# Logistics

- **What you should do...**

- **Project #2** – due Monday, 9/15/25, at 9pm
- **Reading:** zyBooks, Chap 3
- **ZY-3** – due Sunday, 9/14, at 11:59pm
- **Auxiliary Reading:** Lafore, Chap 5 (optional)

- **On your radar:**

- Exam #1 next Wednesday, 9/17, in class. We will discuss it in today's lecture.

# Exam #1

- Wednesday, 9/17
  - Exam must be taken during your assigned class period
- Regular class time (in class, 50 minutes)
  - Arrive early!!!

# Exam #1 – Learning Objectives

- Everything since the beginning of class (8/20) through Friday's lecture (9/12)
- This corresponds to lectures 1-11
- This includes Chapters 1-3 and applicable information from the Java reference chapters 15-21 in zyBooks.
- Programming assignments project #0 & #1

# Exam #1 – Learning Objectives

- **Java Fundamentals (pre-req)**

- Demonstrate proficiency in basic Java syntax, data types, objects (e.g., Strings, arrays, BigNum), concepts and application of OOP
- Perform and implement common operations on arrays, including perfect-sized, oversized, fixed-size, and dynamic arrays.

- **Unit Testing and JUnit**

- Understand the principles of unit testing and its role in software development.
- Implement tests using JUnit to verify the correctness of Java code.

- **Algorithm Analysis**

- Analyze the efficiency of algorithms using Big-O notation.
- Evaluate worst-case, average-case, and best-case performance of algorithms.

# Exam #1 – Learning Objectives

- **Sorting Algorithms**

- Demonstrate knowledge of common sorting algorithms: selection sort, insertion sort, and bubble sort.
- Compare their efficiency and suitability for different use cases.

- **Searching Algorithms**

- Demonstrate knowledge of sequential search and binary search algorithms.
- Compare their efficiency and suitability for different use cases.

- **Linked Lists**

- Understand the structure of linked lists.
- Perform and implement the basic operations of linked lists (e.g., traversal, deletion, and insertion).
- Compare linked lists with arrays, discussing their pros and cons, and use cases.

# Exam #1 – Format

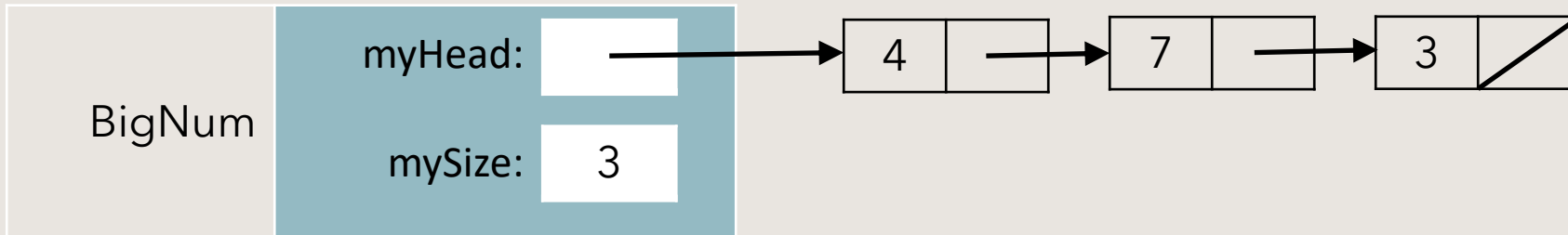
- Some short answer
  - Multiple choice, fill-in-the-blank, T/F
- Some code tracing
  - What does this print out?
  - What's wrong with this?
  - How much work does this do?
- Some code writing
  - Fill in the blank coding
  - Short code segments (e.g., function header, etc.)
  - Small methods or functions
- You will not have a syntax reference sheet

# Programming with Linked Lists

- We often need to process the contents of the list
- This requires walking the list and visiting each node
  - Saw an example earlier to print each value
  - Saw others: convert a BigInt to a printable string and comparing two BigInt objects
- We also need the ability to
  - **Delete** node(s) from a linked list
  - **Insert** node(s) into a linked list

# Deleting a Node

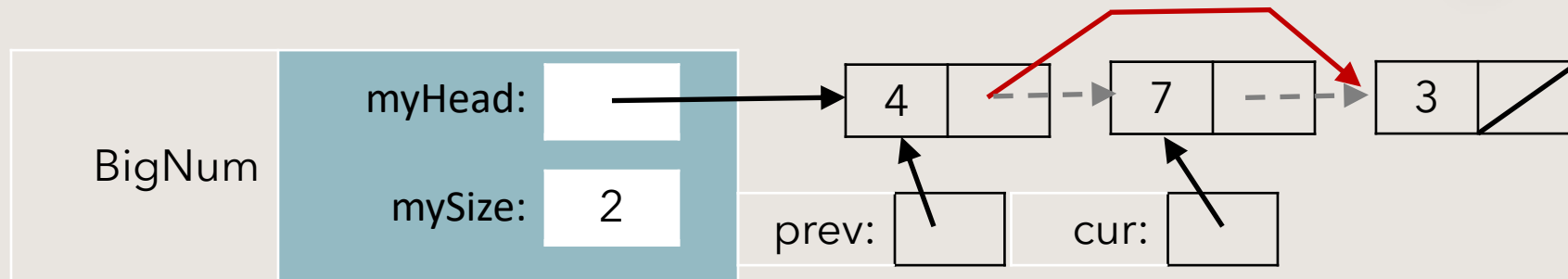
- How about deleting a node from a list?
  - Consider deleting the node holding 7...



- What needs to be done?

# Deleting a Node

- We need the previous node to **link around** the node to be deleted
- Need a `prev` to point to previous node and a `cur` to point to node being deleted



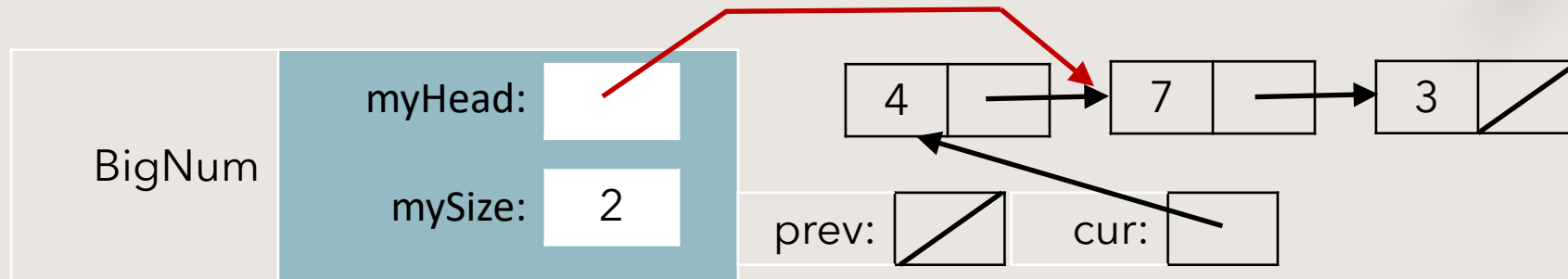
`prev.next = cur.next;`

# Deleting a Node – Edge Conditions

- Deleting the first node
- Deleting the last node
- Deleting the only node

# Deleting the First Node

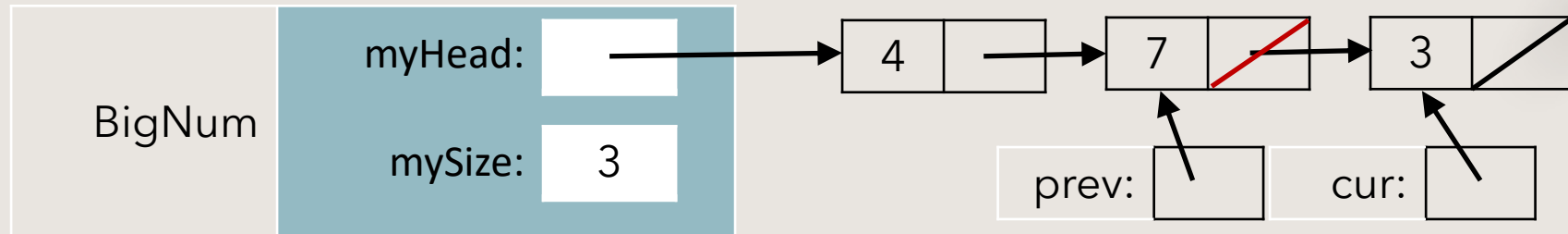
- We need to change `myHead` rather than some node's `next` field
  - Note: there is no previous node, so `prev` is `null`



`myHead = cur.next;`

# Deleting the Last Node

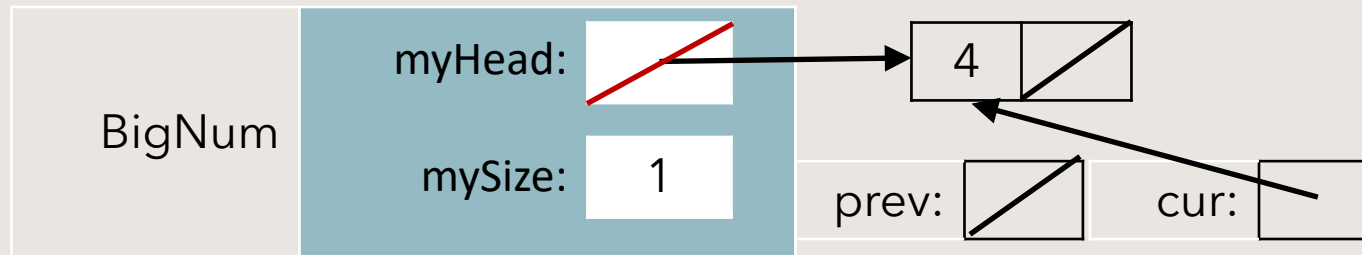
- Same as the general case



`prev.next = cur.next;`

# Deleting the Only Node

- Since the only node must also be the first node, this case is also already handled correctly



`myHead = cur.next;`

# Deleting a Node

- Here's the code to delete a node, **given** that cur references the node to be deleted

```
if (myHead == cur) {           // deleting first node?
    myHead = cur.next;        // unlink the cur node
} else {
    Node prev;                // find previous node
    for (prev = myHead; prev.next != cur; prev = prev.next)
        { /* traverse till the given cur */ }
    prev.next = cur.next;     // unlink the cur node
}

mySize--;
```

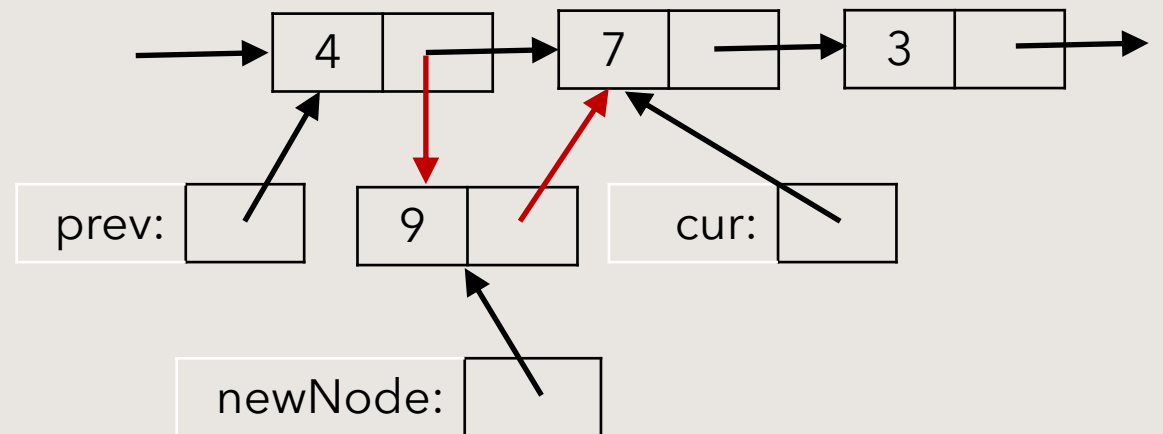
# TopHat



# Inserting a New Node

- Let's assume we know the insertion point, and it is between two nodes: **prev** and **cur**
  - Create the new node and link it in...
  - Must have the previous node point to the new node
  - Must have the new node point to the cur node

```
prev.next = newNode;  
newNode.next = cur;
```

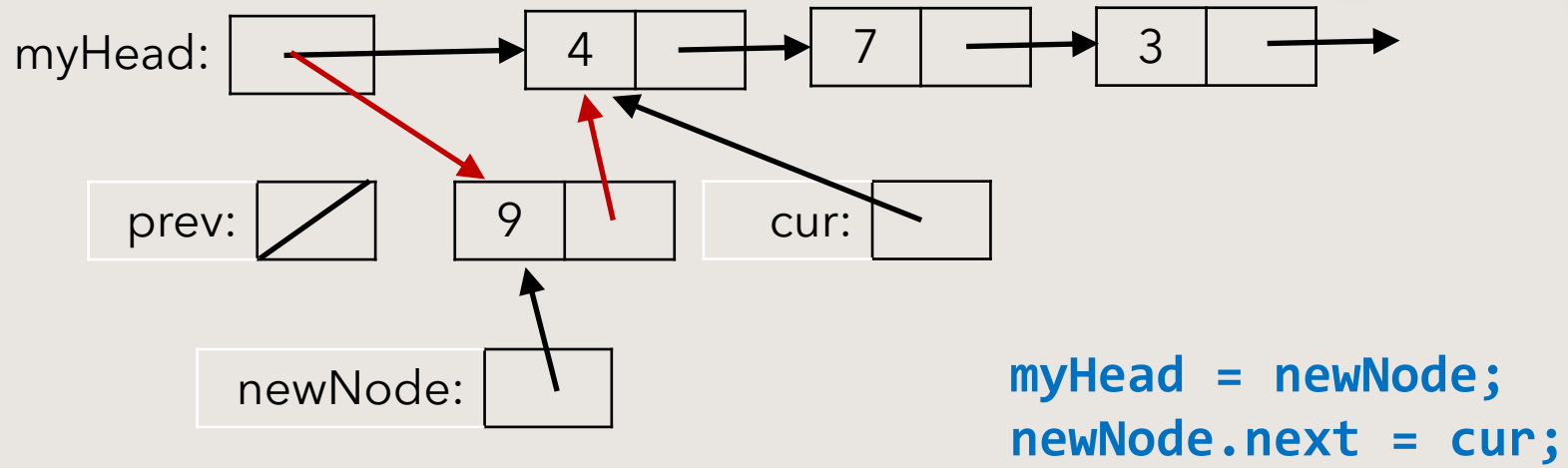


# Inserting a New Node – Edge Conditions

- Inserting at the beginning of the list
- Inserting at the end of the list
- Inserting into an empty list

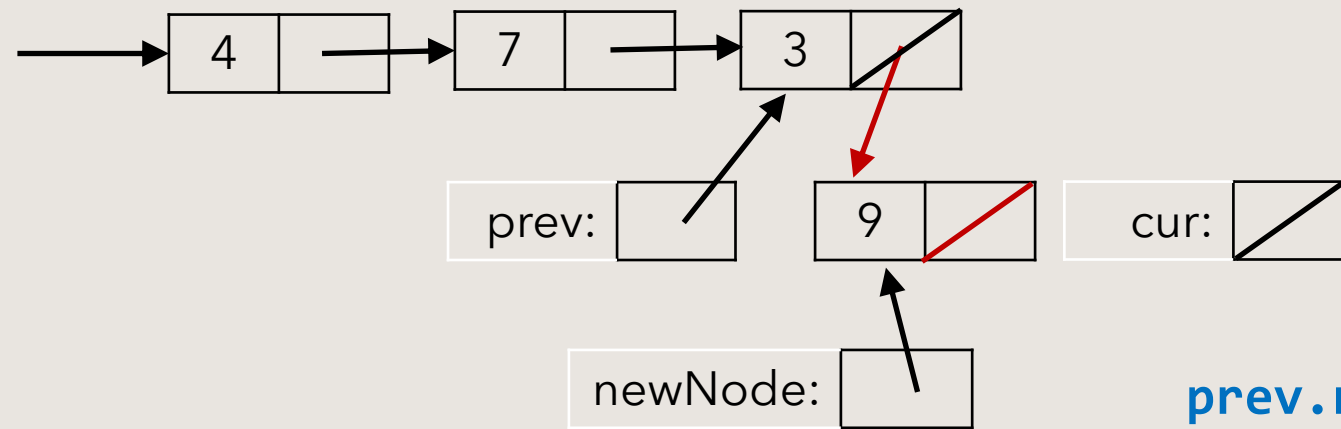
# Inserting at the Beginning of the List

- There is no previous node, so prev is null
- Must update myHead rather than a node's next field



# Inserting at the End of the List

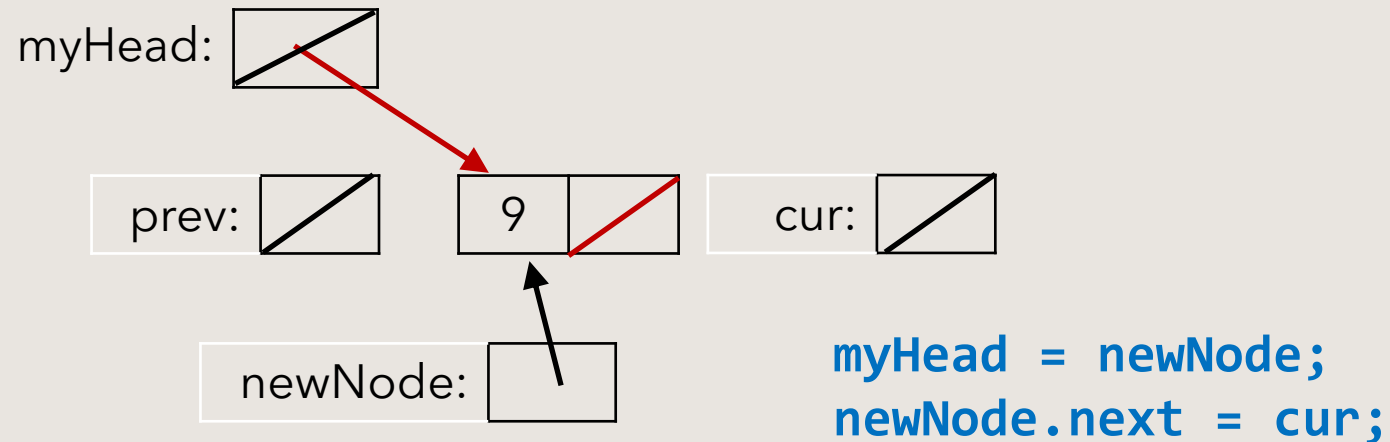
- There is no following node, so `cur` is null
- `newNode.next` still gets assigned the value of `cur`
  - So not really a special case at all



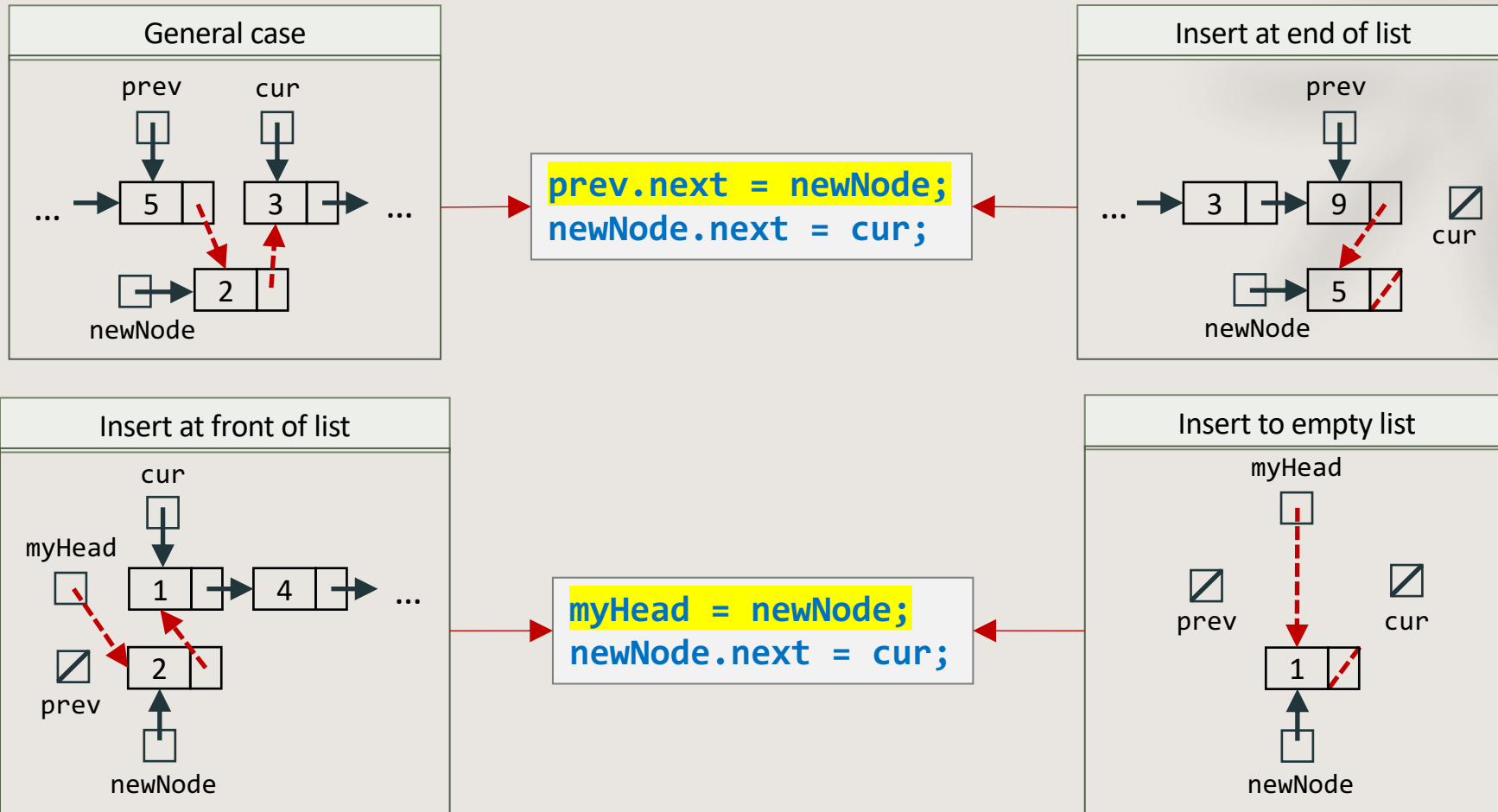
```
prev.next = newNode;  
newNode.next = cur;
```

# Inserting into an Empty List

- In this case, both `prev` and `cur` will be null
- Just like adding to the front of the list, we must update `myHead` rather than a node's `next` field
  - So this is already handled by our earlier edge condition



# Inserting a New Node – Summary



# Inserting a New Node

- So, how do we determine prev and cur?
- We must walk the list until we find the insertion point
- Initially, we start a traversal of the list:  
    prev = null;  
    cur = myHead;
- Next, there must be some criteria to determine the insertion point
  - E.g., for a sorted list, find the first node with a value larger than the new value
  - We simply traverse the list until the criteria is satisfied **and** we haven't fallen off the end of the list

# Inserting a New Node

- Code to traverse the list:

```
while (cur != null && criteriaTest(newNode, cur)) {  
    prev = cur;  
    cur = cur.next;  
}
```

- E.g., for a sorted list, the criteria test is simply:  
newNode.data > cur.data
- Is the order of boolean expressions important?
  - YES!!
  - Need to make sure that cur is not null before you dereference it

# Inserting a New Node – while Loop

```
// Pre: newNode refers to node we want to insert into the list
Node prev = null;
Node cur = myHead;

while (cur != null && criteriaTest(newNode, cur)) {
    prev = cur;
    cur = cur.next;
}
// now that prev & cur are set, do insertion
if (prev == null) {
    myHead = newNode;
} else {
    prev.next = newNode;
}
newNode.next = cur;

mySize++;
```

# Inserting a New Node

Does this code handle all our edge conditions?

1. Does it handle an empty list?

- YES!
- Both prev and cur will be null

2. Does it handle inserting at the front of the list?

- YES!
- prev will be null and cur will be the same as myHead

3. Does it handle inserting at the end of the list?

- YES!
- prev will point to the last node and cur will be null

```
// Pre: newNode refers to node
// we want to insert into the list
Node prev = null;
Node cur = myHead;

while (cur != null && criteriaTest(newNode, cur)) {
    prev = cur;
    cur = cur.next;
}
// now that prev & cur are set, do insertion
if (prev == null) {
    myHead = newNode;
} else {
    prev.next = newNode;
}
newNode.next = cur;

mySize++;
```

# Inserting a New Node – for Loop

```
// Pre: newNode refers to node we want to insert into the list
Node prev, cur;
for (prev = null, cur = myHead;
     cur != null && criteriaTest(newNode, cur);
     prev = cur, cur = cur.next) {
    // nothing to do
}
// now that prev & cur are set, do insertion
if (prev == null) {
    myHead = newNode;
} else {
    prev.next = newNode;
}
newNode.next = cur;

mySize++;
```