COSC 121: Computer Programming II

Dr. Bowen Hui
University of British Columbia
Okanagan

Abstract Data Types

- An abstract data type (ADT) is an organized collection of information
  - Provides a set of operations to manage that info
  - These operations define the interface to the ADT
- Purpose:
  - Lets other classes use ADT based on operations expected
  - No need to know how operations are implemented
  - Provides layer of abstraction in the design
- E.g., car ADT
  - Start the engine, change gears, drive, turn, accelerate, stop, etc.

Data Structures

- Data types in Java: int, double, String, etc.
- Data structure we've seen: array
- Array is a static data structure because it has a fixed size
- E.g., `int[] intarray = new int[ 10 ];`
  - Length of this array is always 10
  - What if we don't need to use all the slots?
  - What if we need to use more slots?
- BUT: Sometimes you want to have a data structure that can change in size

How to Implement?

- Example:

```
int[] intarray = new int[ 10 ];
intarray[0] = 0;
intarray[1] = 1;
. . .
intarray[9] = 9;
```

- What if you want to continue adding elements?

How to Implement?

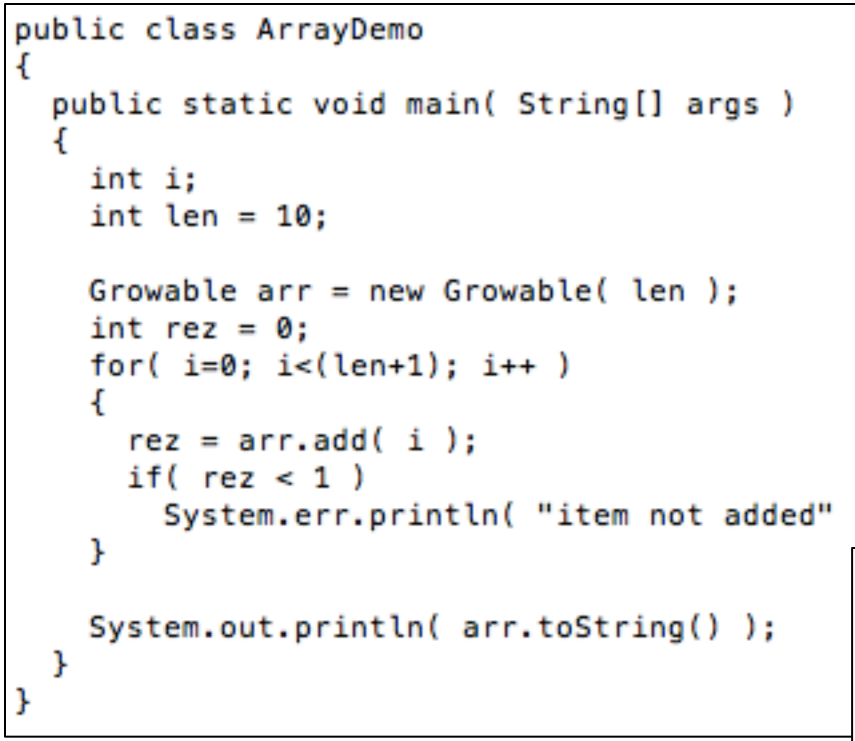
- Example:

```
int[] intarray = new int[ 10 ];
intarray[0] = 0;
intarray[1] = 1;
. . .
intarray[9] = 9;
```

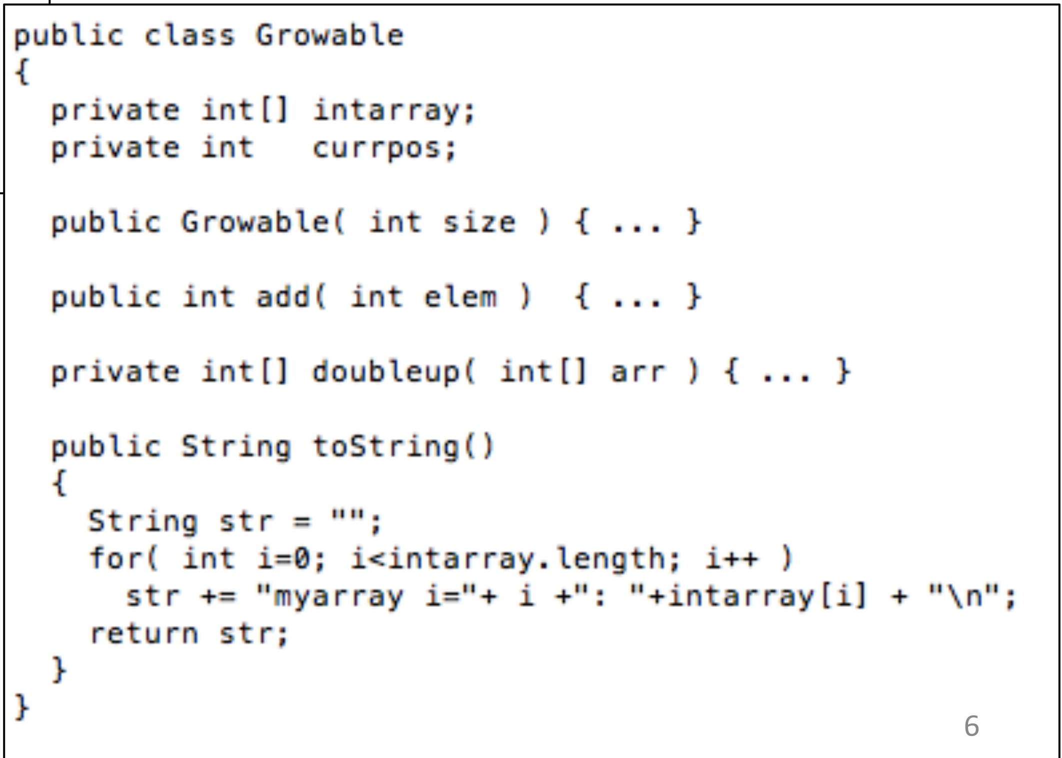
- What if you want to continue adding elements?
  - Double the size of the array as needed
  - Create a new array
  - Copy all the old elements into it
  - Lots more room to add additional elements

Exercise

- Complete the `Growable` class:
  - `Growable` initializes variables
  - `add()` adds an element into currpos of intarray
  - `doubleup()` returns a new integer array that is twice as large



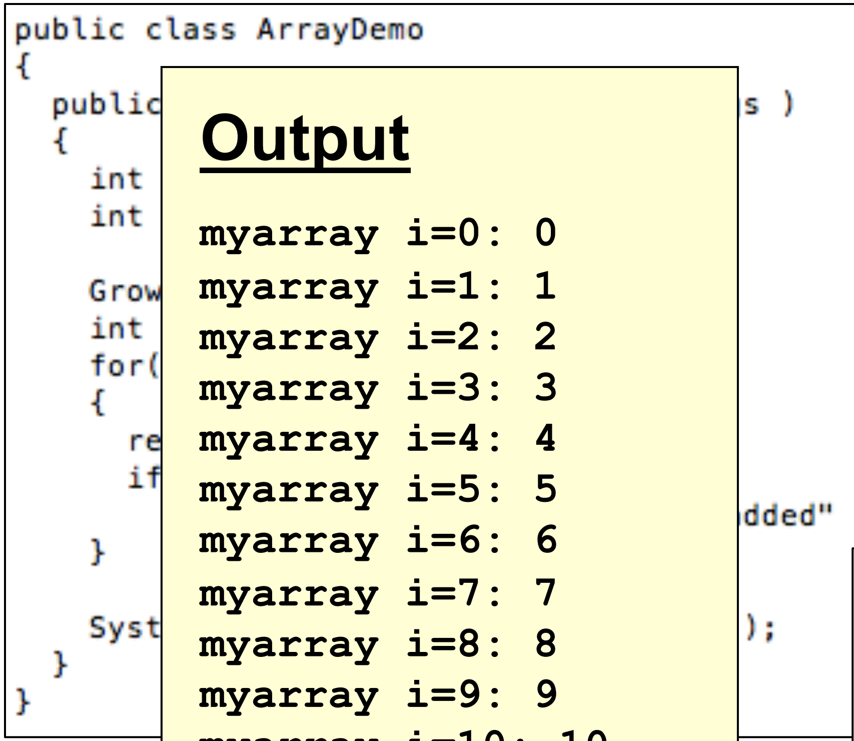
```
public class ArrayDemo
{
  public static void main( String[] args )
  {
    int i;
    int len = 10;

    Growable arr = new Growable( len );
    int rez = 0;
    for( i=0; i<(len+1); i++ )
    {
      rez = arr.add( i );
      if( rez < 1 )
        System.err.println( "item not added" );
    }

    System.out.println( arr.toString() );
  }
}
```



```
public class Growable
{
  private int[] intarray;
  private int   currpos;

  public Growable( int size ) { ... }

  public int add( int elem )  { ... }

  private int[] doubleup( int[] arr ) { ... }

  public String toString()
  {
    String str = "";
    for( int i=0; i<intarray.length; i++ )
      str += "myarray i="+ i +": "+intarray[i] + "\n";
    return str;
  }
}
```

Exercise

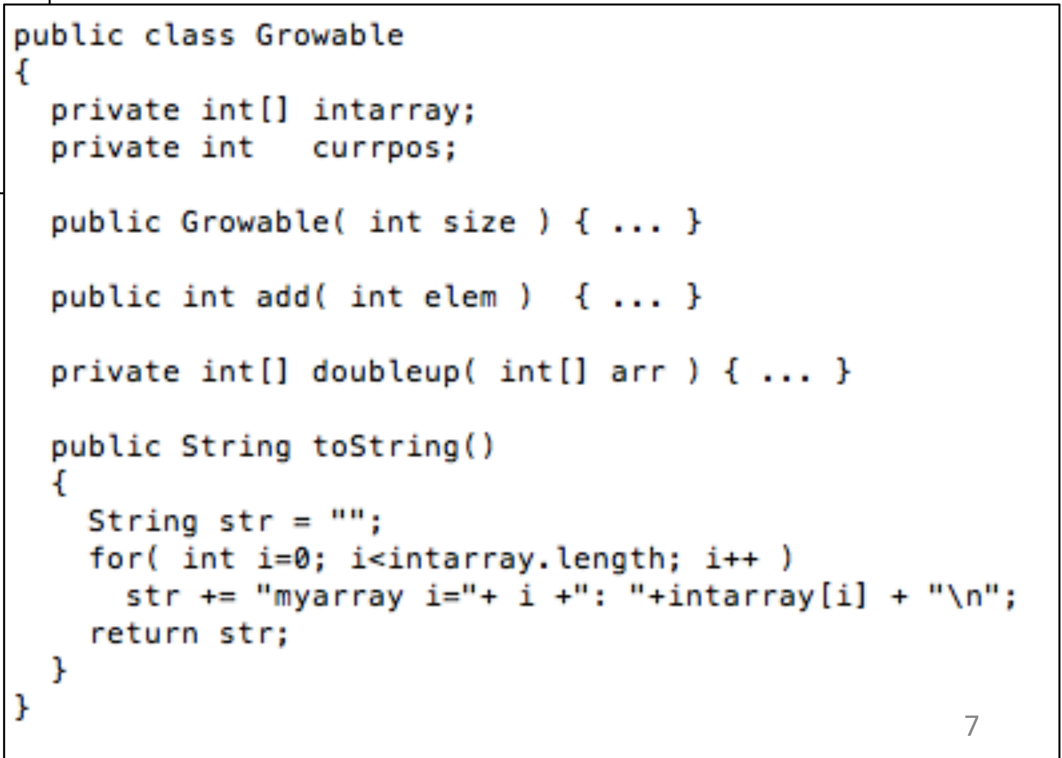
- Complete the `Growable` class:
  - `Growable` initializes variables
  - `add()` adds an element into currpos of intarray
  - `doubleup()` returns a new integer array that is twice as large



```
public class ArrayDemo
{
  public                                            s )
  {
    int
    int

    Grow
    int
    for(
    {
      re
      if
                                                  dded" );
    }

    Syst                                            );
  }
}
```

Output

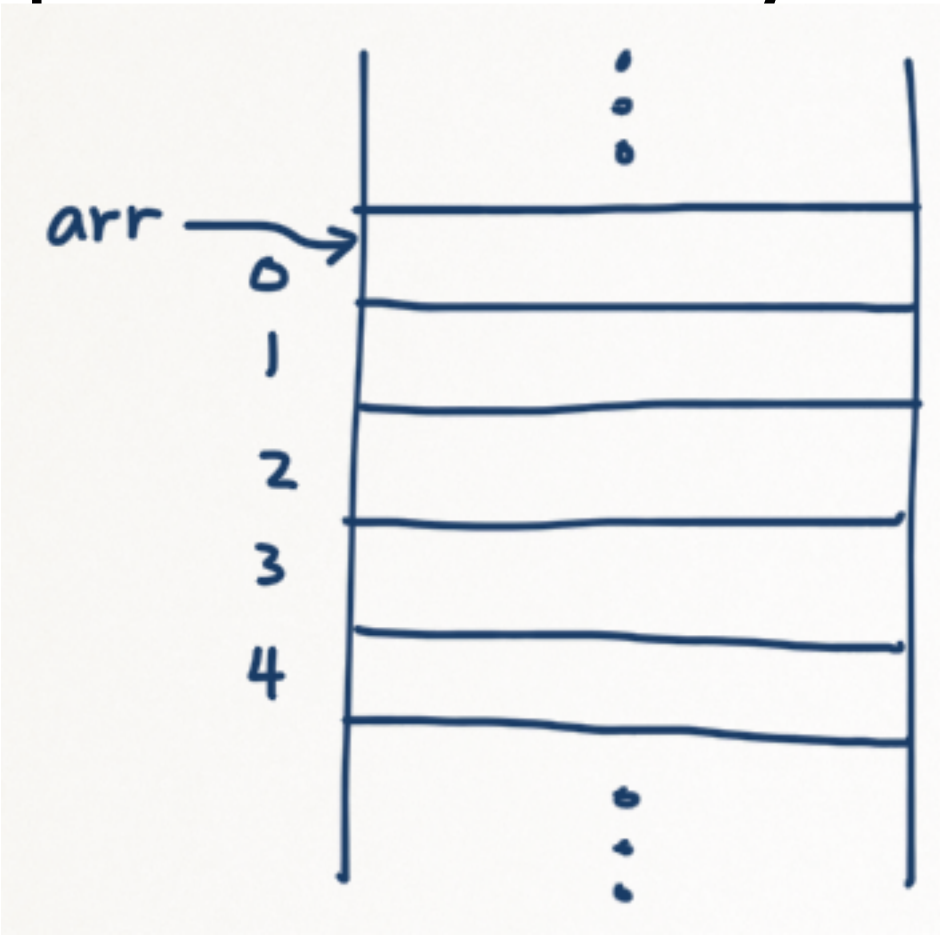
```
myarray i=0:  0
myarray i=1:  1
myarray i=2:  2
myarray i=3:  3
myarray i=4:  4
myarray i=5:  5
myarray i=6:  6
myarray i=7:  7
myarray i=8:  8
myarray i=9:  9
myarray i=10:  10
myarray i=11:  0
myarray i=12:  0
myarray i=13:  0
myarray i=14:  0
myarray i=15:  0
myarray i=16:  0
myarray i=17:  0
myarray i=18:  0
myarray i=19:  0
```



```
public class Growable
{
  private int[] intarray;
  private int   currpos;

  public Growable( int size ) { ... }

  public int add( int elem )  { ... }

  private int[] doubleup( int[] arr ) { ... }

  public String toString()
  {
    String str = "";
    for( int i=0; i<intarray.length; i++ )
      str += "myarray i="+ i +": "+intarray[i] + "\n";
    return str;
  }
}
```

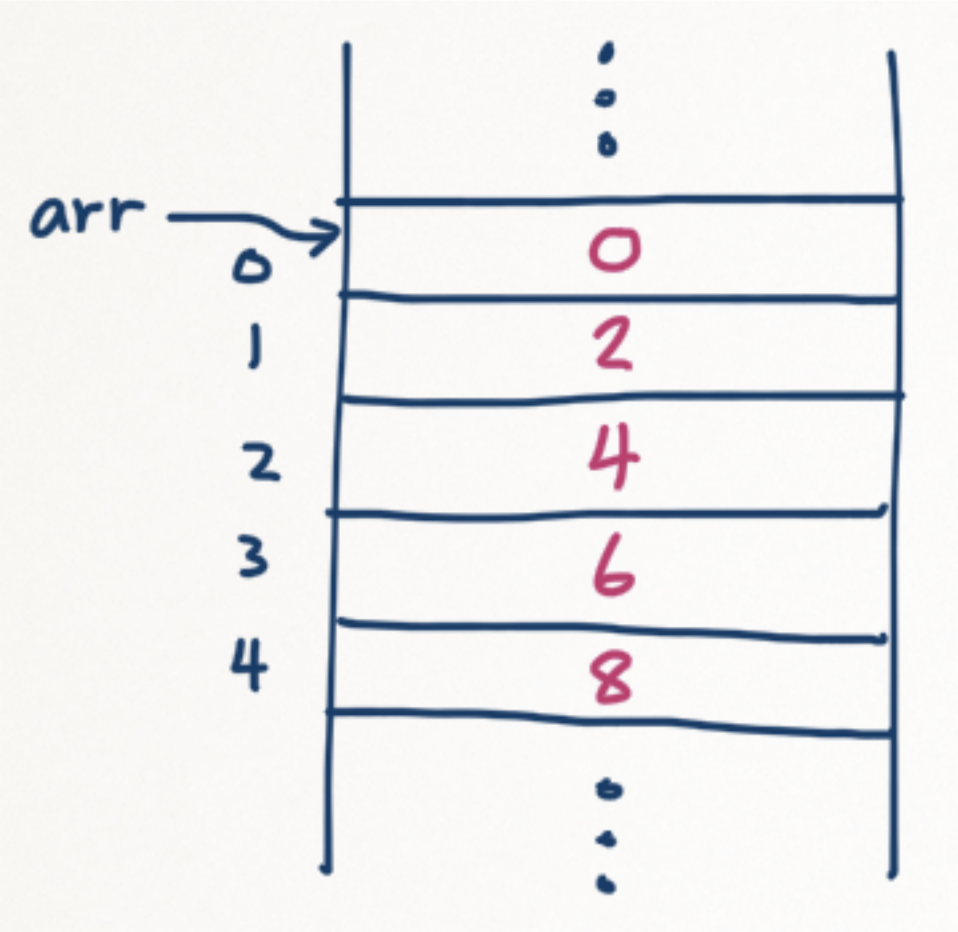
What Happens in Memory

```
int[] arr = new int[5];
```



What Happens in Memory

```
int[] arr = new int[5];
for( int i=0; i<arr.length; i++ )
    arr[i] = 2*i;
// do some stuff
// occupy additional memory
```



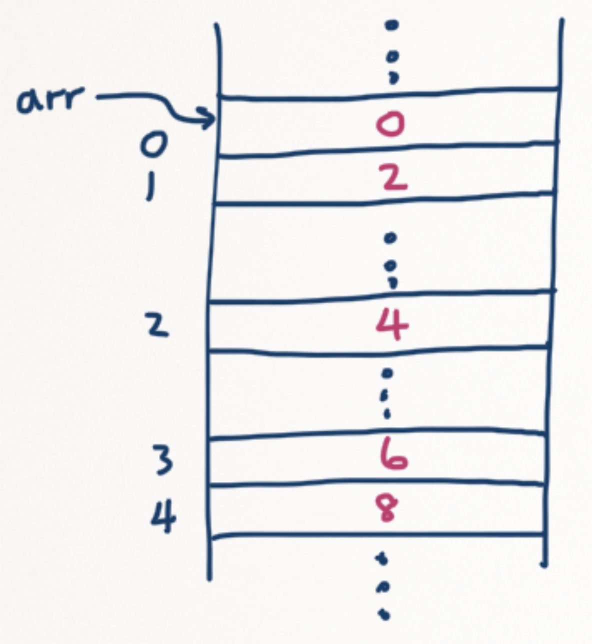
How to Manage Dynamic Structures?

- If array is “growable”:
  - Need to find a contiguous block of memory to create a bigger array
- If array is “shrinkable”:
  - Give up some memory slots that other code may later occupy
- Real problem: How big should we make an array?
  - Memory is not unlimited
  - Finding an arbitrarily large contiguous block of memory is not always feasible

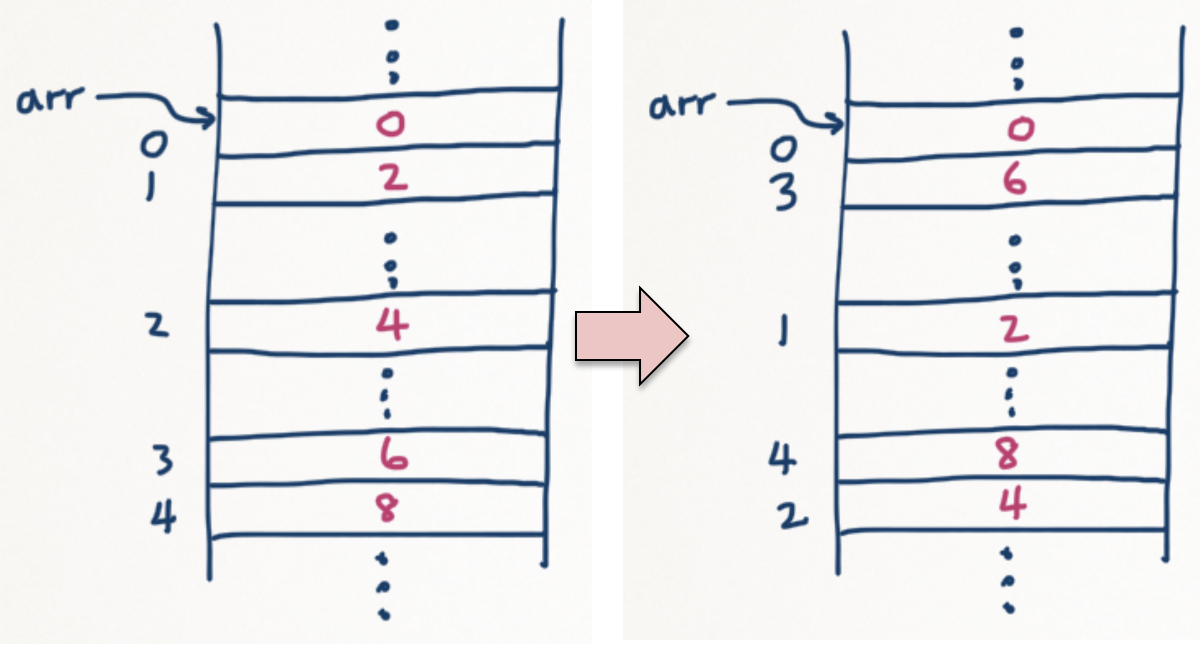
Managing Dynamic Structures

- Solution:
  - Break up what you need to store into small pieces
  - Efficient use of memory
  - Add and delete small memory blocks as we go
- Downside:
  - You need to keep track of the positioning manually

Using Non-Contiguous Memory

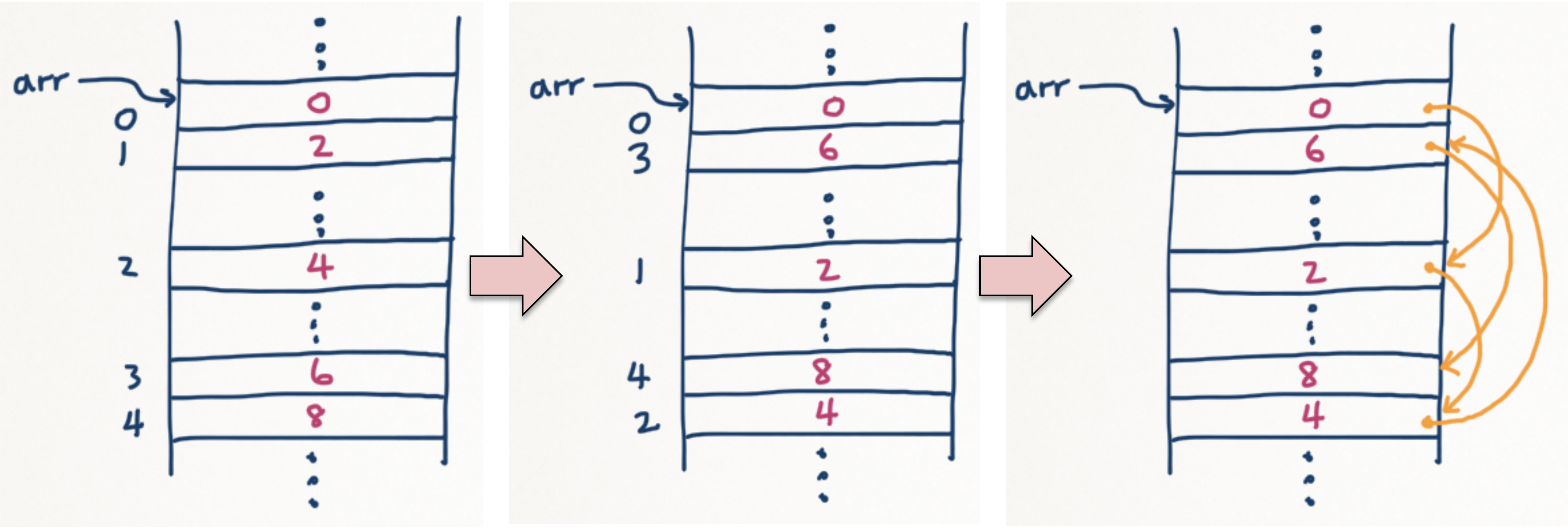


Using Non-Contiguous Memory



indices no longer
in meaningful
order

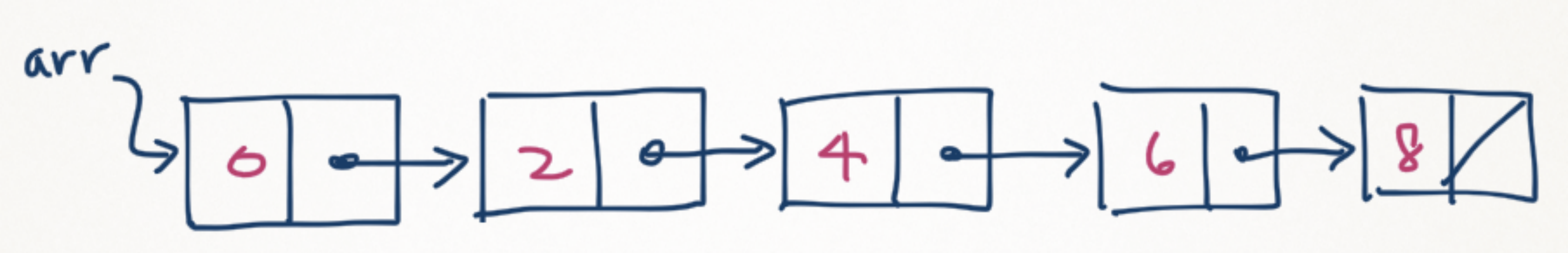
Using Non-Contiguous Memory



indices no longer
in meaningful
order

Simplified Visualization

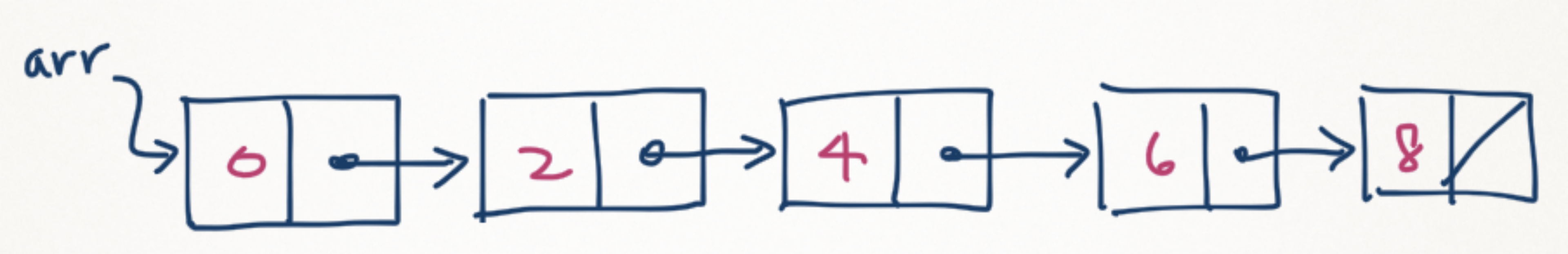
- Visualizing the data in its own context
- This is a linked list representation
- Each node is its own structure



- Managing space for adding and deleting elements is easier

How to Implement a Linked List?

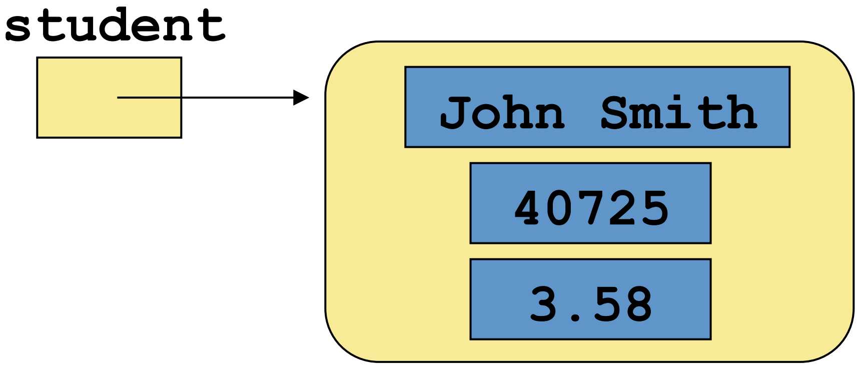
- Recall representation:



- What is a node?
- How to create two nodes?
- How to link one node to another?

Review: Object References

- Object reference is a variable that stores the address of an object
- A reference also can be called a pointer
- References often are depicted graphically:



Implementing Nodes

- Nodes can be created as class objects
- Object references can be used to create links between objects
- E.g.:

```
public class Node
{
   private int info;
   private Node next;
}
```

- A node can hold any kind/quantity of info

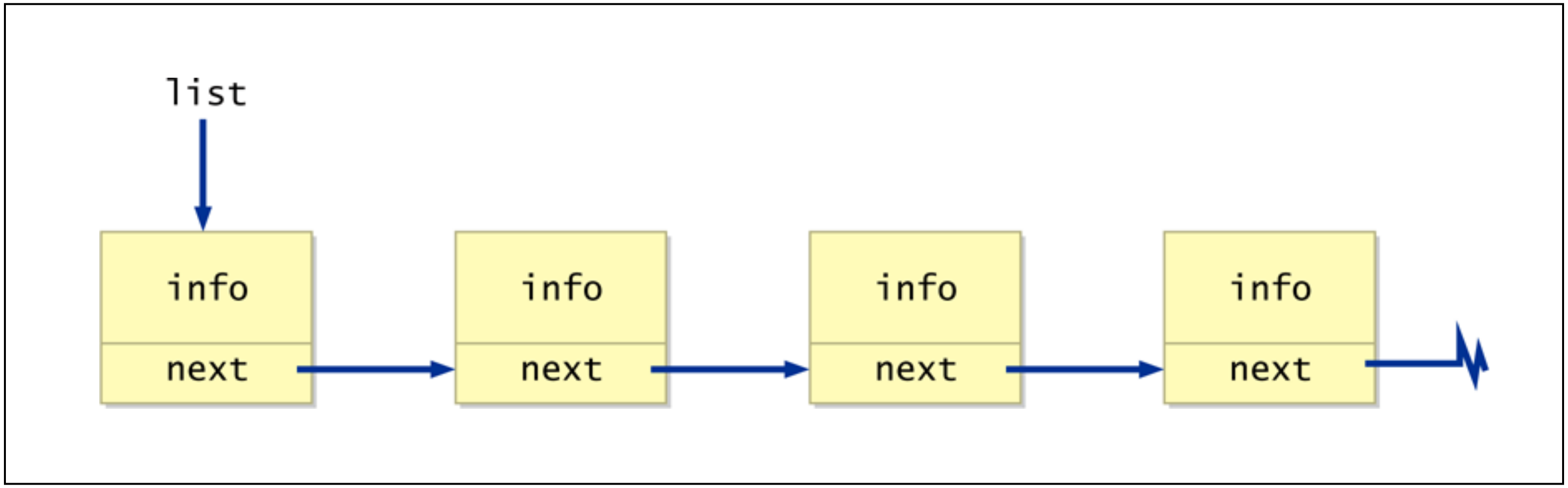
Exercise

- Define a class called StudentNode that holds:
  - A name
  - A student ID
  - And a GPA
- Create two student nodes
- Get the first node to point to the second node
- Recall:

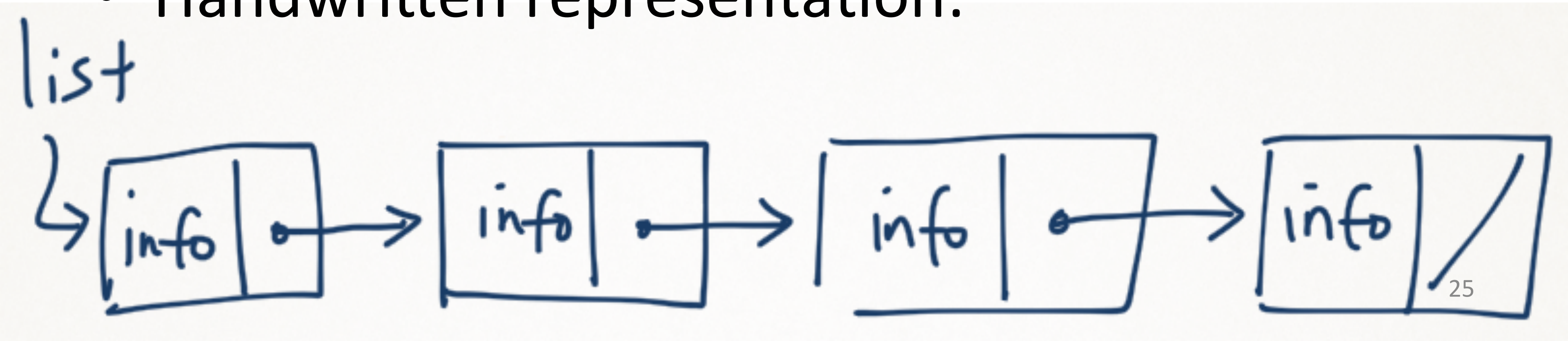
```
public class ExampleNode
{
  private int info;
  private ExampleNode next;
}
```

Visual Representation of Linked List

- Textbook representation:



- Handwritten representation:



Operations to Manipulate a Linked List

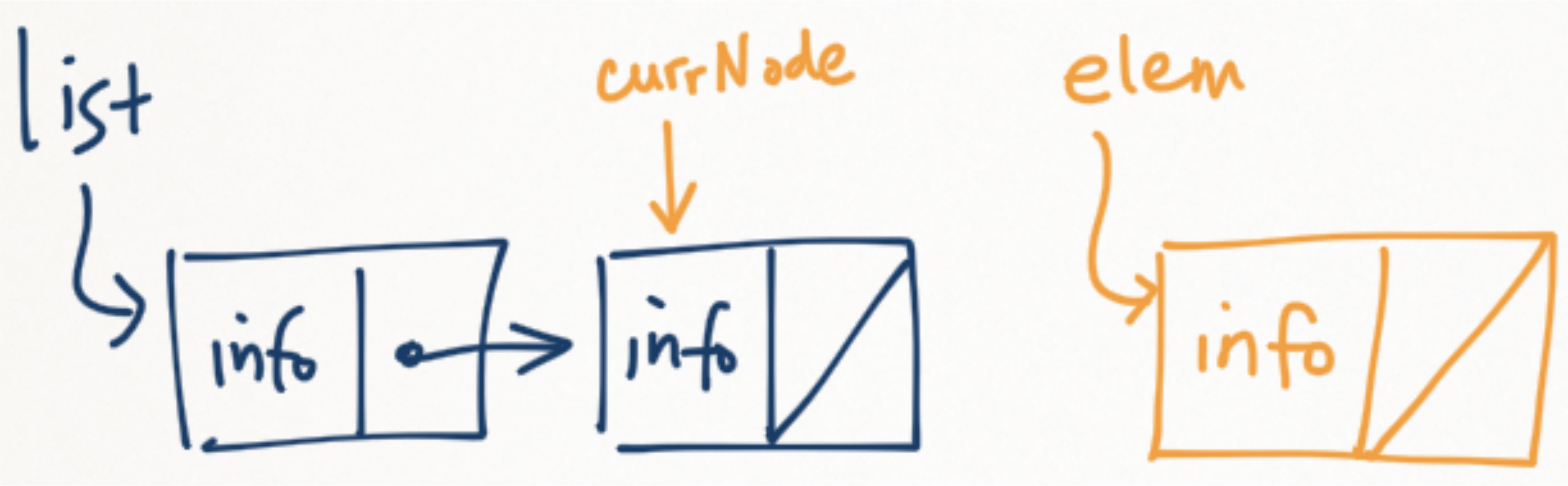
- Add a node
  - Add a new node to the end of the list
- Delete a node
  - Find a node
  - Delete it from the list
  - Reconnect remaining list
- Insert a node
  - Add a new node in a particular location within the list

Operations to Manipulate a Linked List (cont.)

- Advanced operations:
  - Search (needed for delete, insert)
  - Sort (facilitates other operations)
- Search for a node
  - Find a node with certain information in it
- Sort the entire linked list
  - Based on the information stored in the nodes, reorder the list

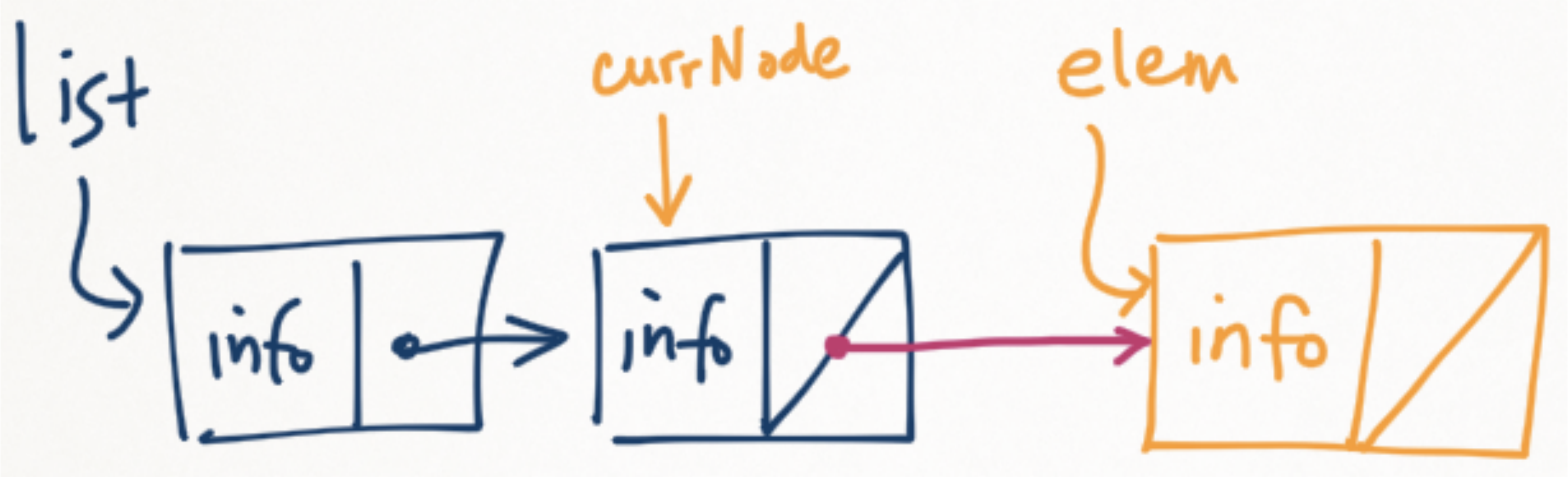
Adding a Node

- A node can be added to the end of a linked list by changing the `next` pointer of the preceding node:



Adding a Node

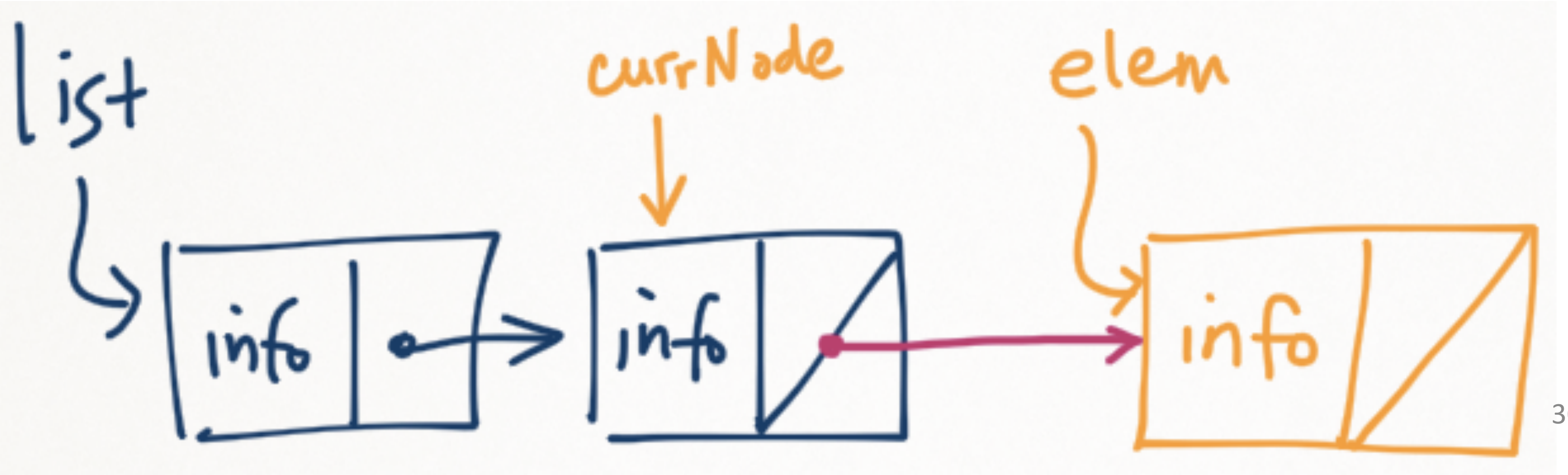
- A node can be added to the end of a linked list by changing the `next` pointer of the preceding node:



Quick Check

Write code that adds `elem` after the node pointed to by `currNode`.

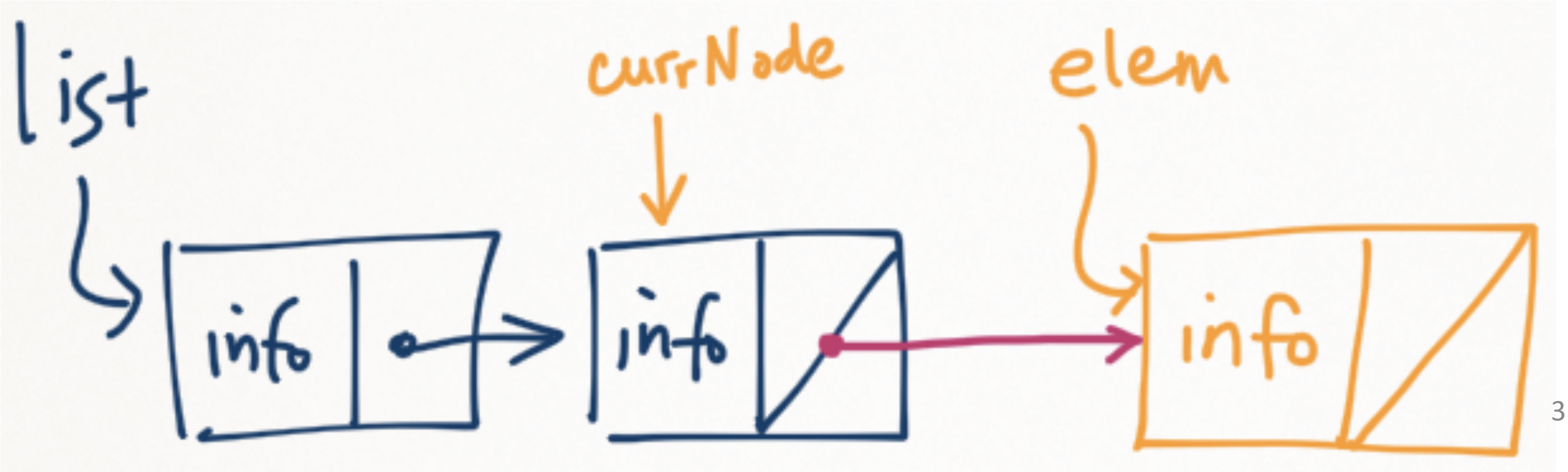
```
Node elem = new Node( info );
// get currNode to point to elem
```



Quick Check

Write code that adds `elem` after the node pointed to by `currNode`.

```
Node elem = new Node( info );
currNode.next = elem;
```



Quick Check

What if the list is empty? Write code that adds `elem` to an empty list.

Quick Check

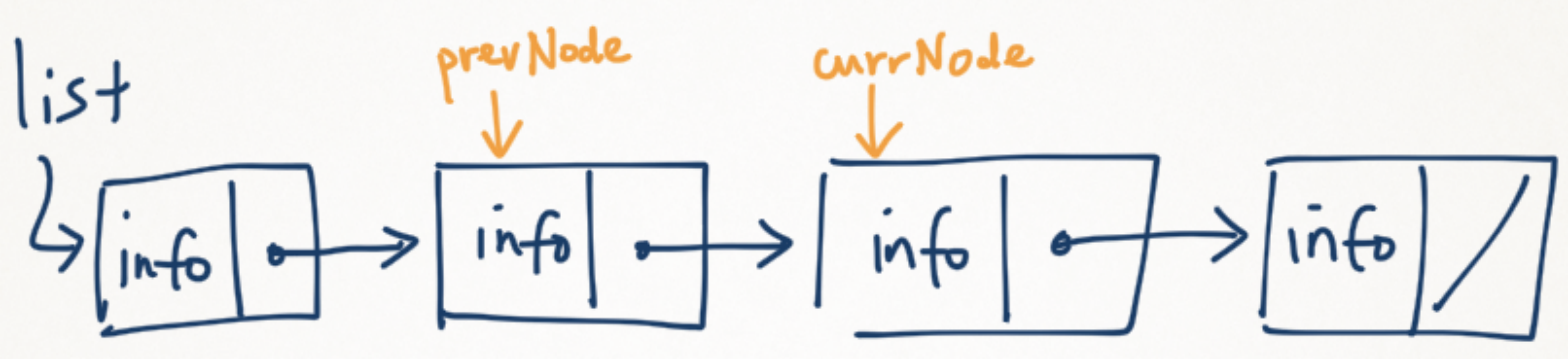
What if the list is empty? Write code that adds `elem` to an empty list.

```
Node elem = new Node( info );
list = elem;
```

What is `list`?

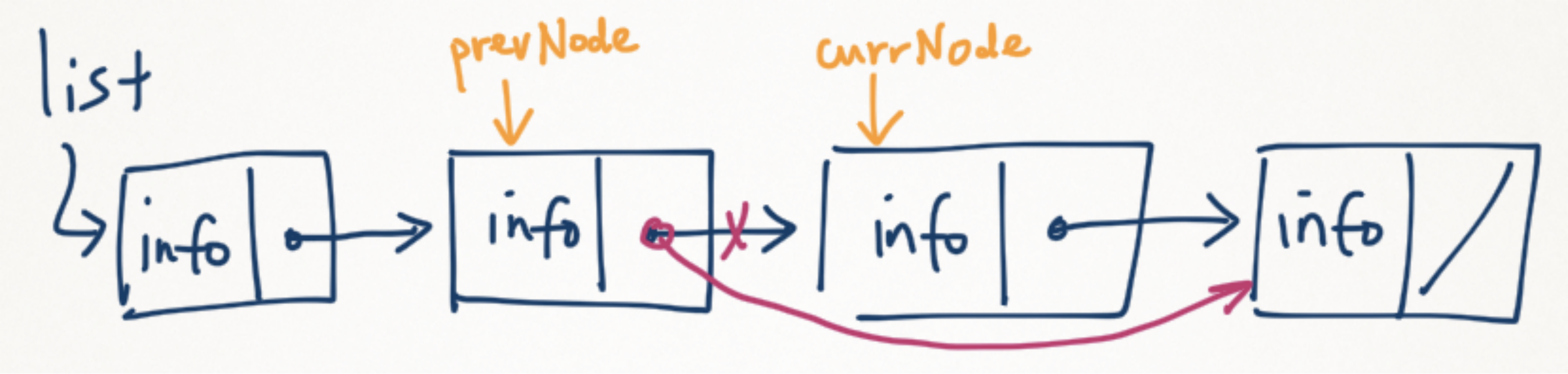
Deleting a Node

- Likewise, a node can be removed from a linked list by changing the `next` pointer of the preceding node:



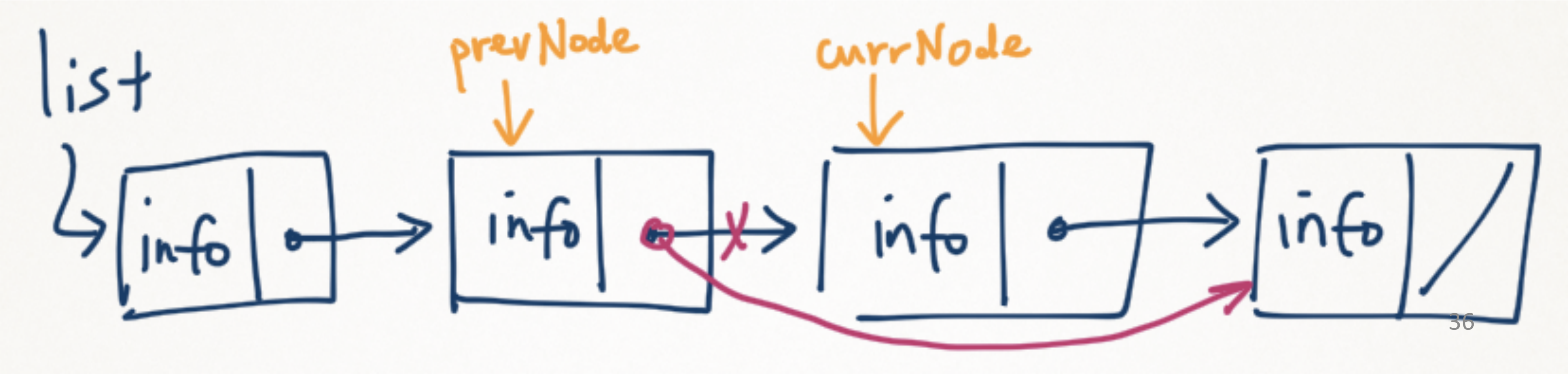
Deleting a Node

- Likewise, a node can be removed from a linked list by changing the `next` pointer of the preceding node:



Quick Check

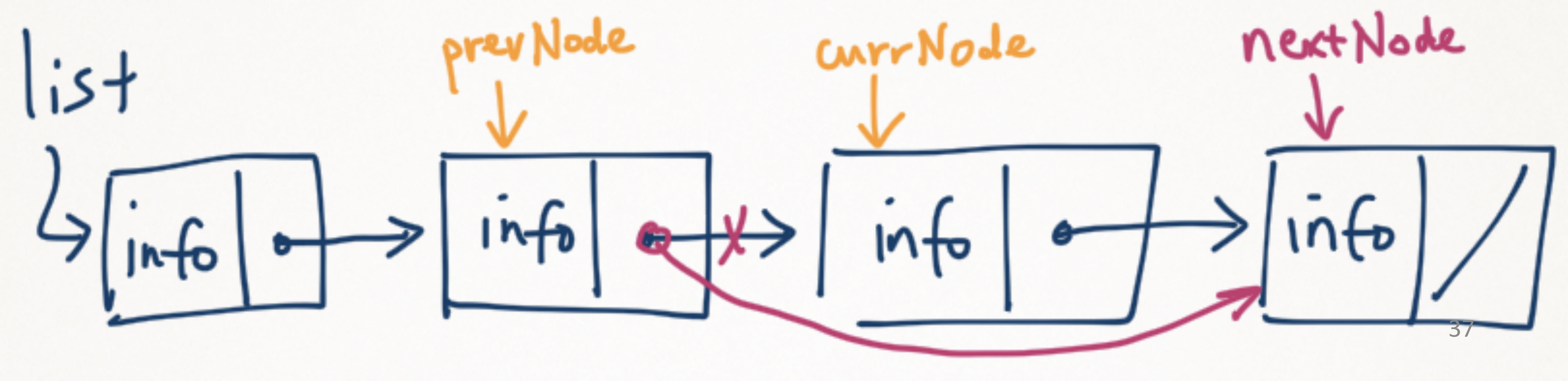
Write code that deletes `currNode` from the list.



Quick Check

Write code that deletes `currNode` from the list.

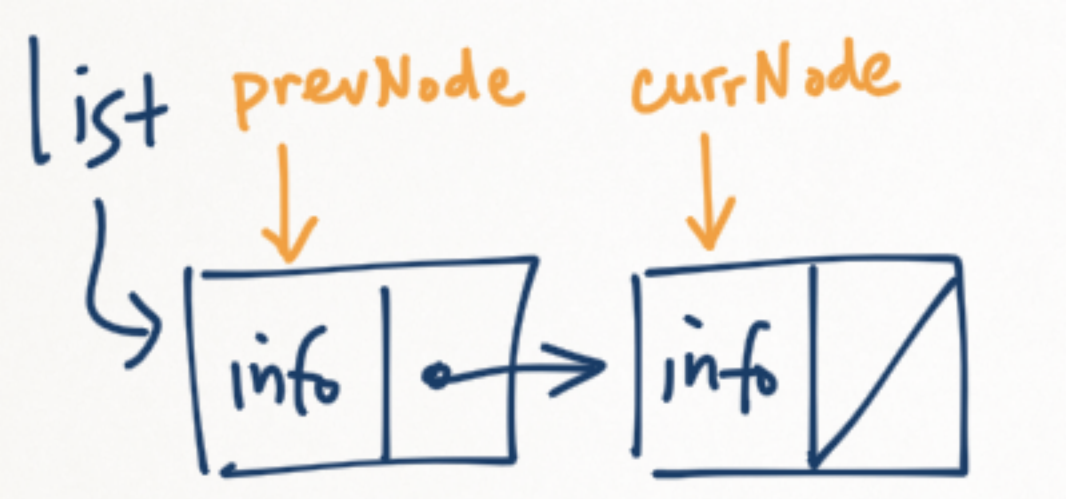
```
// reconnect prevNode to nextNode
Node nextNode = currNode.next;
prevNode.next = nextNode;
```



Quick Check

What if the node to delete is …

- At the end of the list?

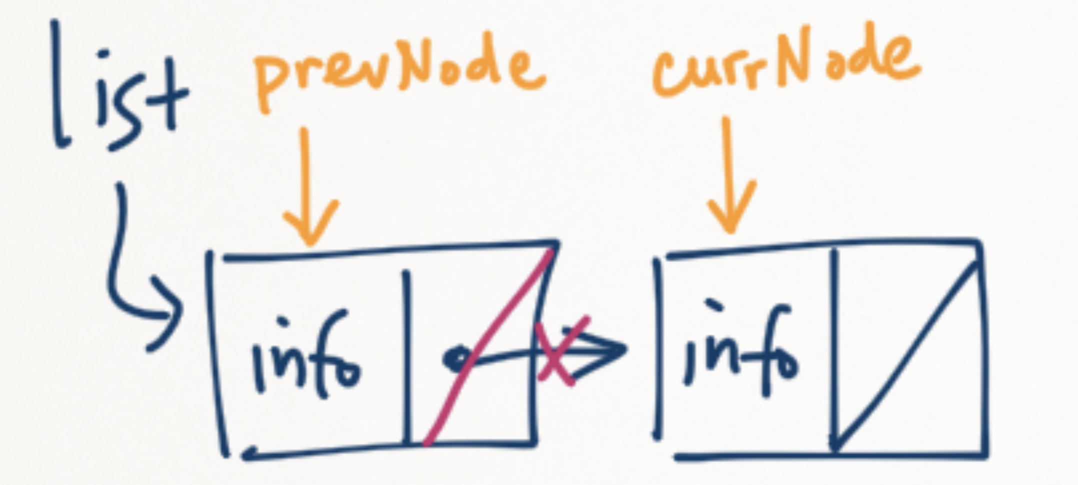


Quick Check

What if the node to delete is …

- At the end of the list?

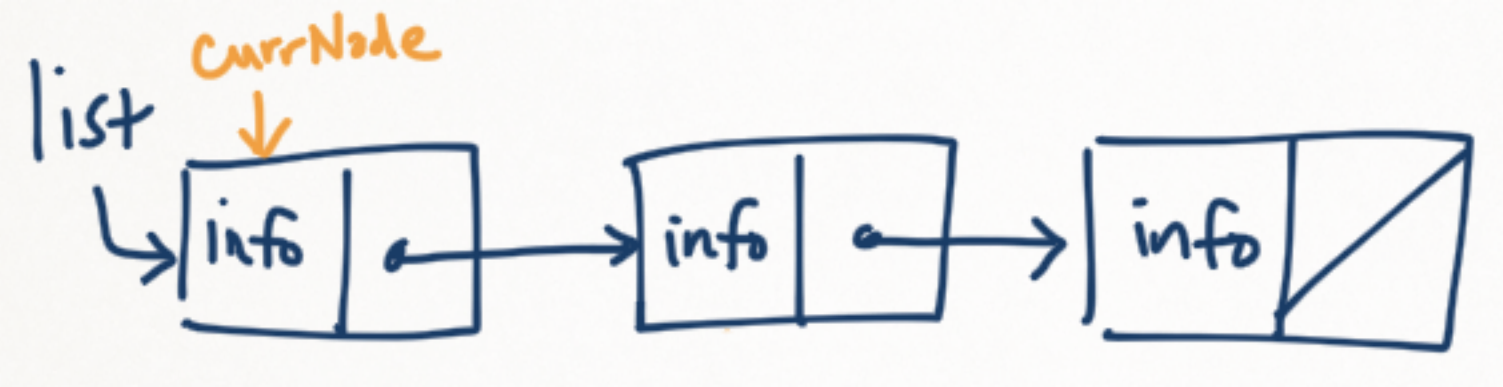
```
// end
prevNode.next = null;
```



Quick Check

What if the node to delete is …

- At the beginning (first node) of the list?

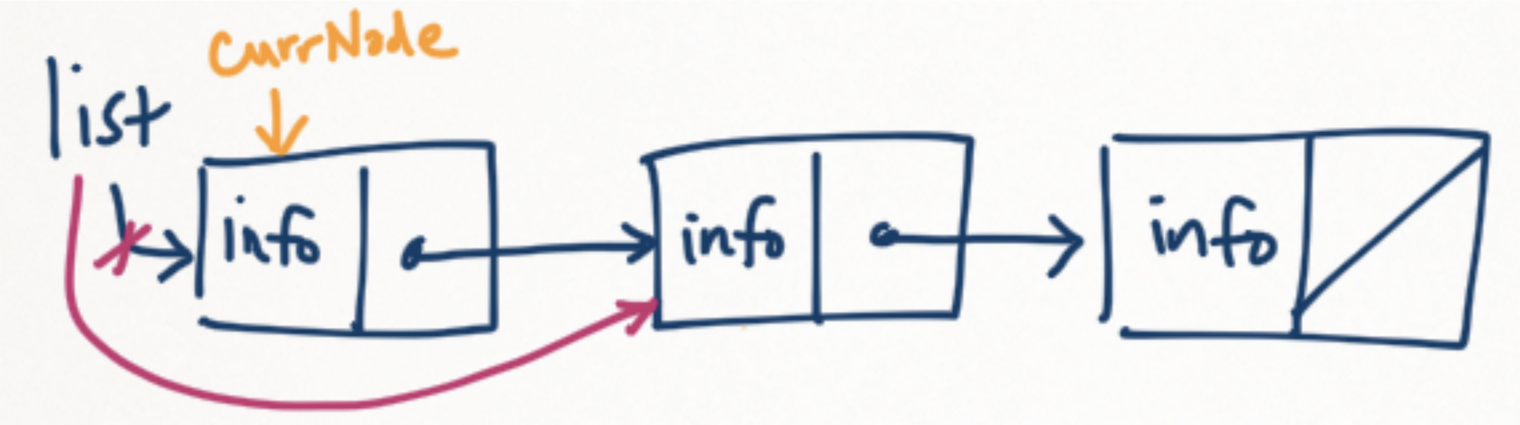


Quick Check

What if the node to delete is …

- At the beginning (first node) of the list?

```
// beginning
list = list.next;
// may update currNode too
```



Quick Check

What if the node to delete is …

- Not found?

Quick Check

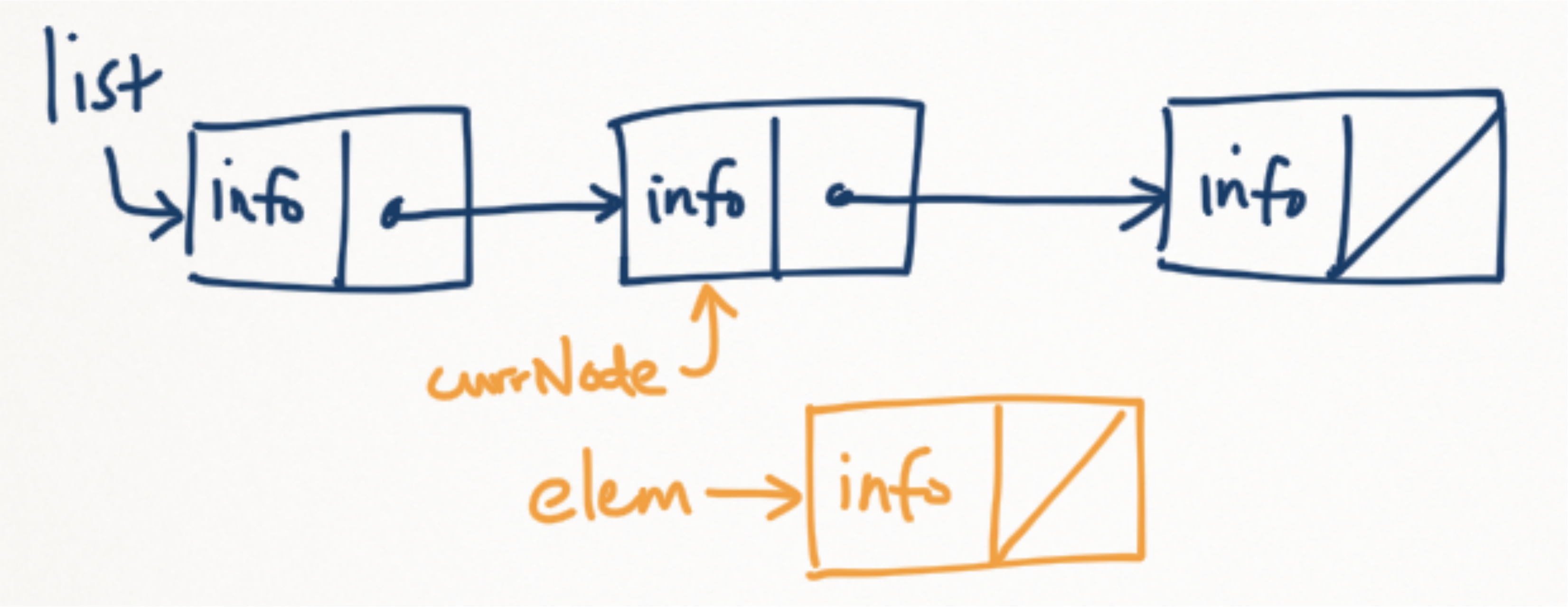
What if the node to delete is …

- Not found?

```
// do nothing
```

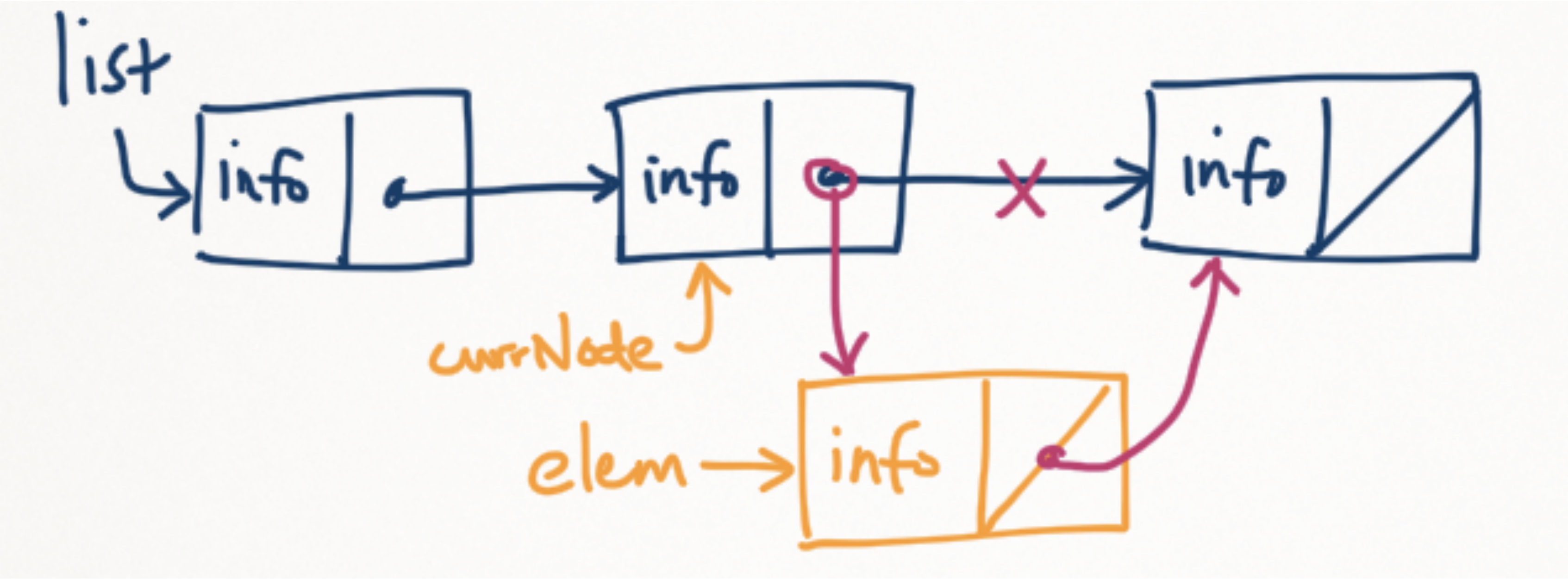
Inserting a Node

- A node can be inserted into a linked list with a few reference changes:



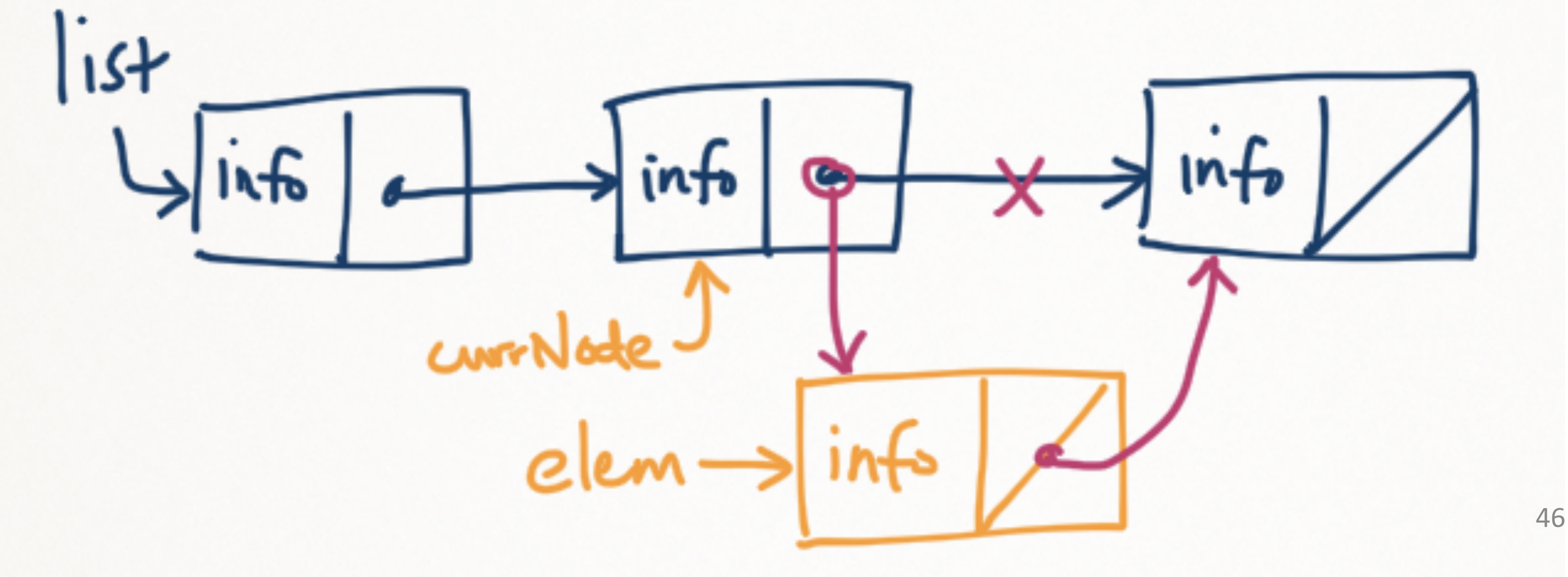
Inserting a Node

- A node can be inserted into a linked list with a few reference changes:



Quick Check

Write code that inserts `elem` after the node pointed to by `currNode`.

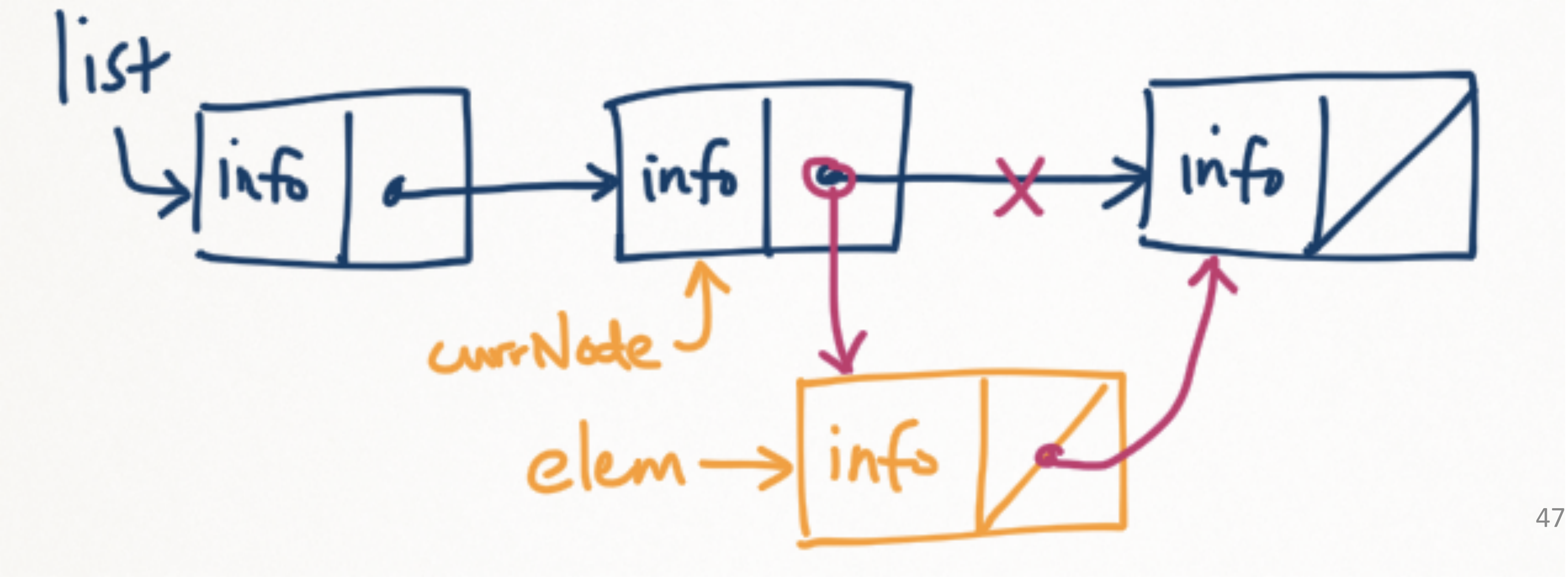


Quick Check

Write code that inserts `elem` after the node pointed to by `currNode`.

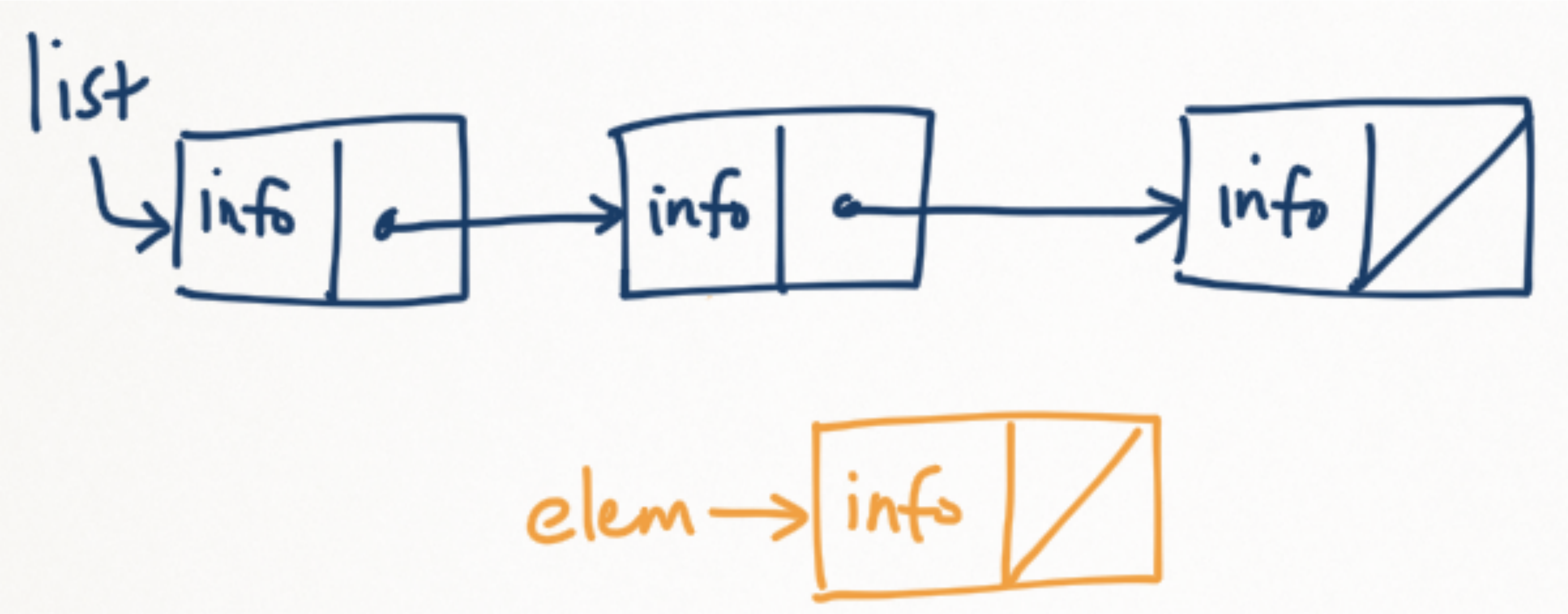
```
elem.next = currNode.next;

currNode.next = elem;
```

} reversible?



Quick Check

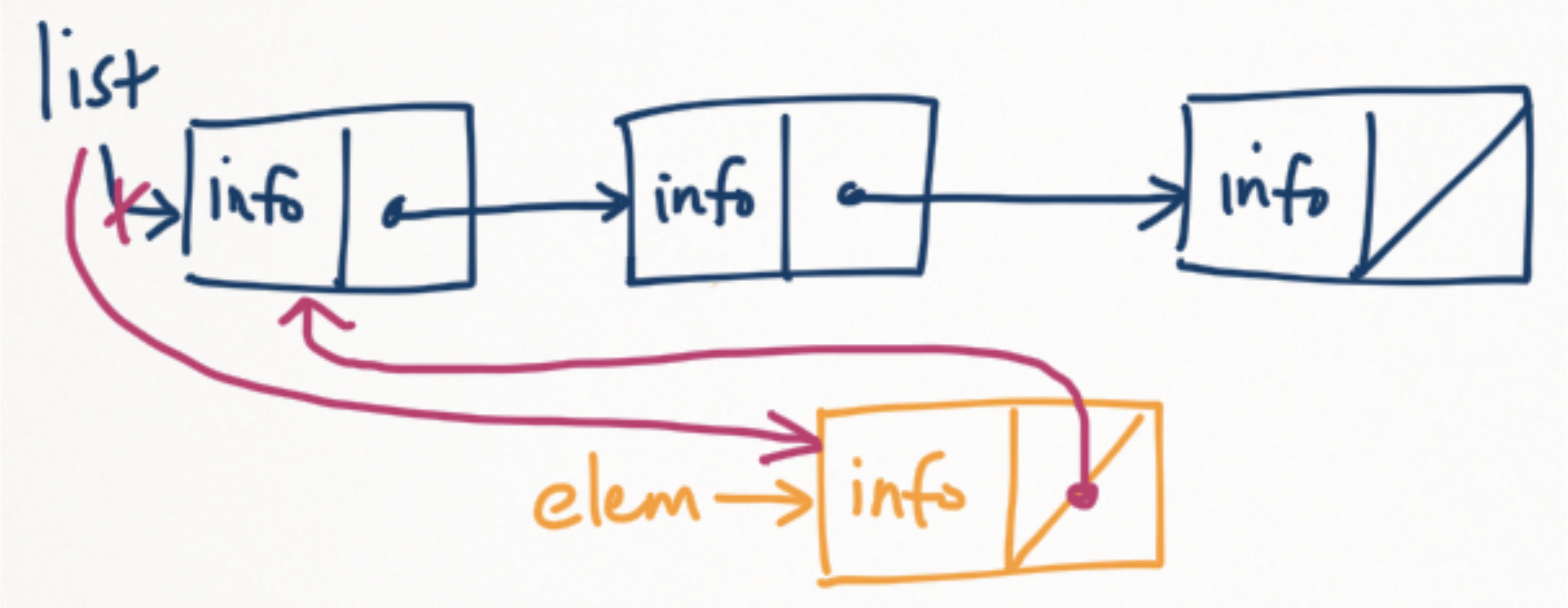
Write code that inserts `elem` as the first element in the list.



Quick Check

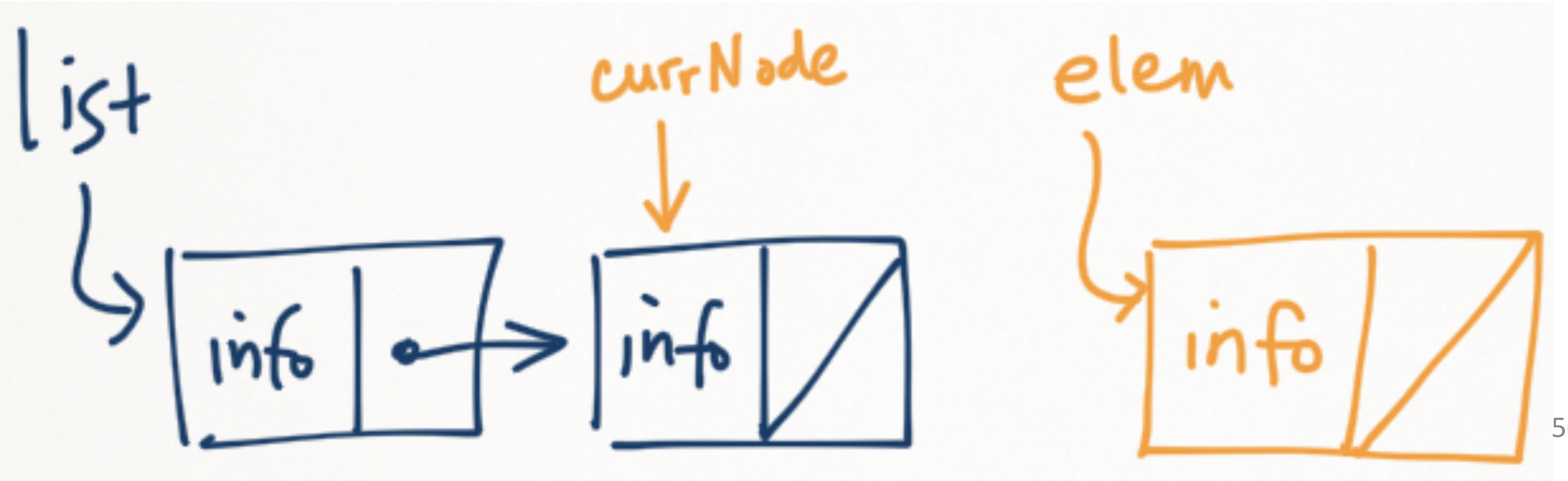
Write code that inserts `elem` as the first element in the list.

```
elem.next = list;
list = elem;
```



Quick Check

Write code that inserts `elem` as the last element in the list.

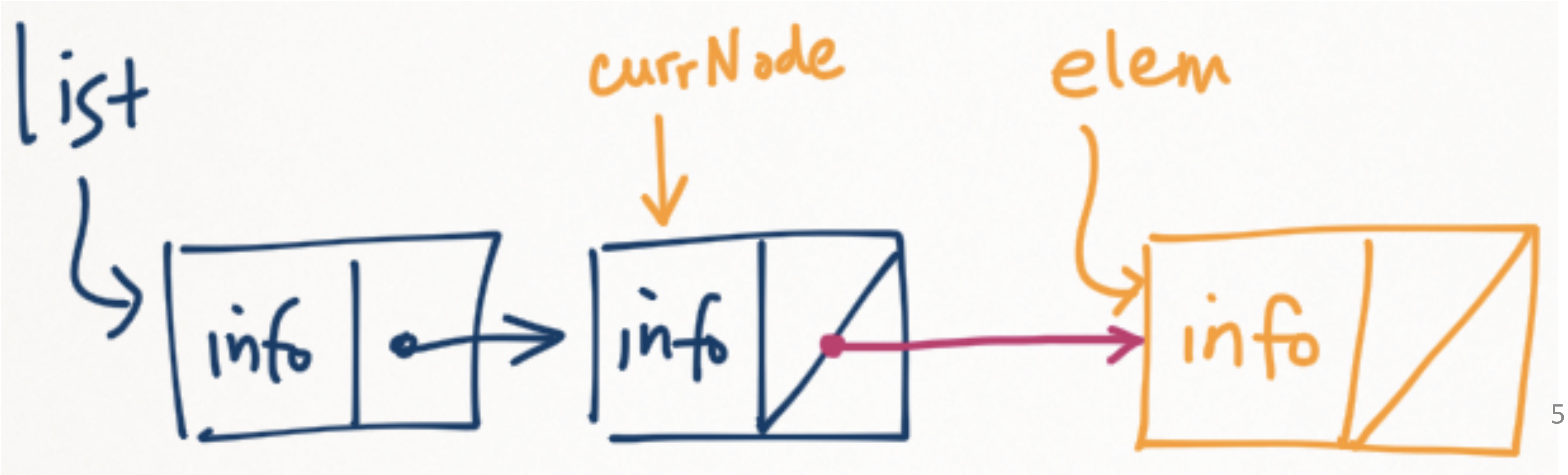


Quick Check

Write code that inserts `elem` as the last element in the list.

```
currNode.next = elem;
```

What does this remind you of?



Summary of ADTs

- An abstract data type (ADT) is an organized collection of information
- Makes use of OOP technique called abstraction
- Static lists store information in fixed sized structures
  - Easier to manage and operate
  - Less efficient in memory use
- Dynamic lists store information in variable sized structures
  - Efficient memory usage
  - Harder to implement (must manage everything manually)
  - Abstracts implementation detail from other classes