CSCI 232:
Data Structures and Algorithms

Linked Lists, Stacks, Queues

Reese Pearsall

Spring 2025

Announcements

Our TAs for CSCI 232

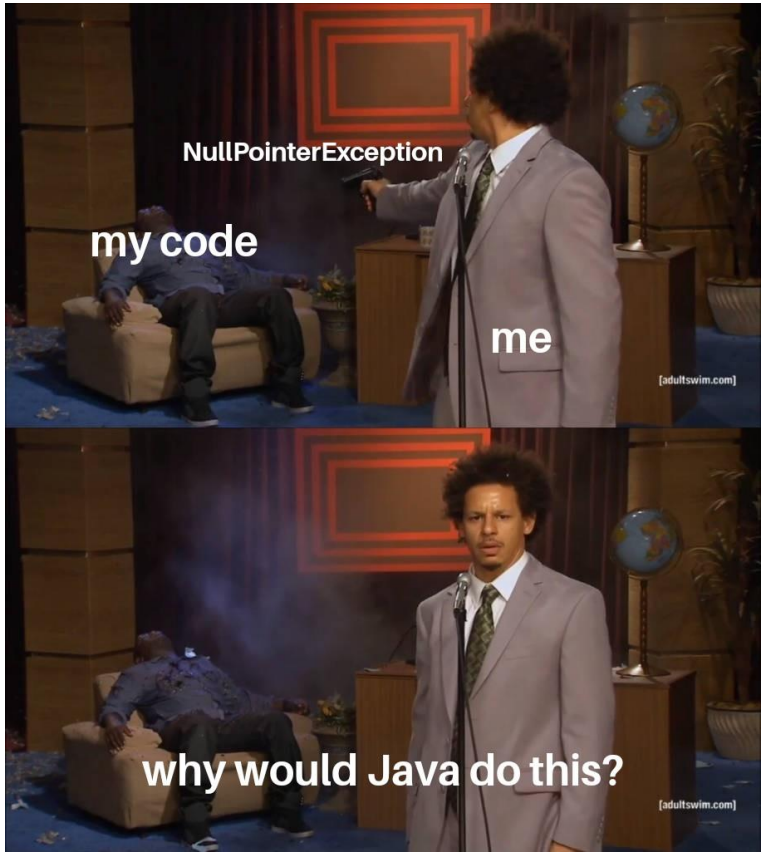
TA Office Hours are in Barnard Hall 259

Section 003 (Friday 10-12)
- **Shahnaj Mou**
- shahnajmou@gmail.com
- Office Hours: Mondays 3:10 PM – 4:10 PM
  5:10 PM - 6:00 PM

Section 004 (Friday 12-2)
- **Oscar Oropeza**
- ooropeza2000@gmail.com
- Office Hours: Wednesday 12PM – 1PM

Section 005 (Friday 2-4)
- **Shahnaj Mou**
- shahnajmou@gmail.com
- Office Hours: Mondays 3:10 PM – 4:10 PM
  5:10 PM - 6:00 PM

First lab is posted and due on Friday!



Reese's Office hours are in Roberts Hall 111 today

An **array** is a fixed-sized, linear collection of elements

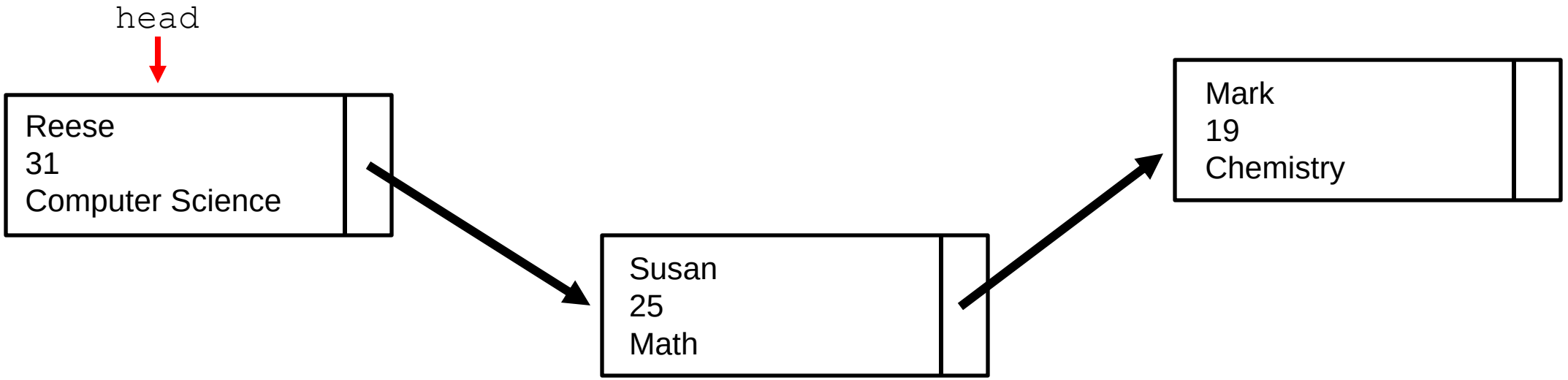
You can use the built-in Java array

A **list** is a dynamic, linear collection of elements

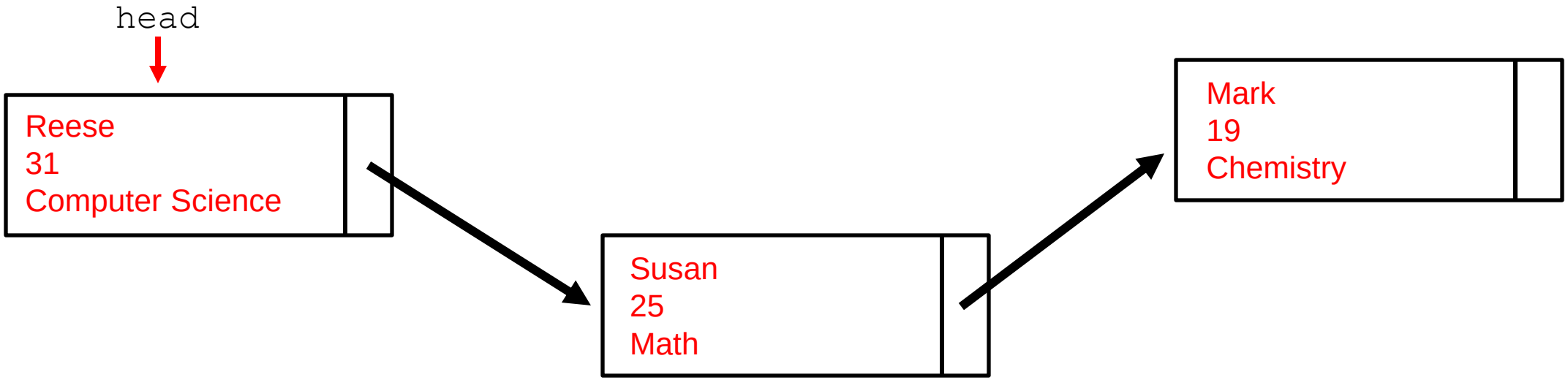
`ArrayList<E>`

`LinkedList<E>`

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



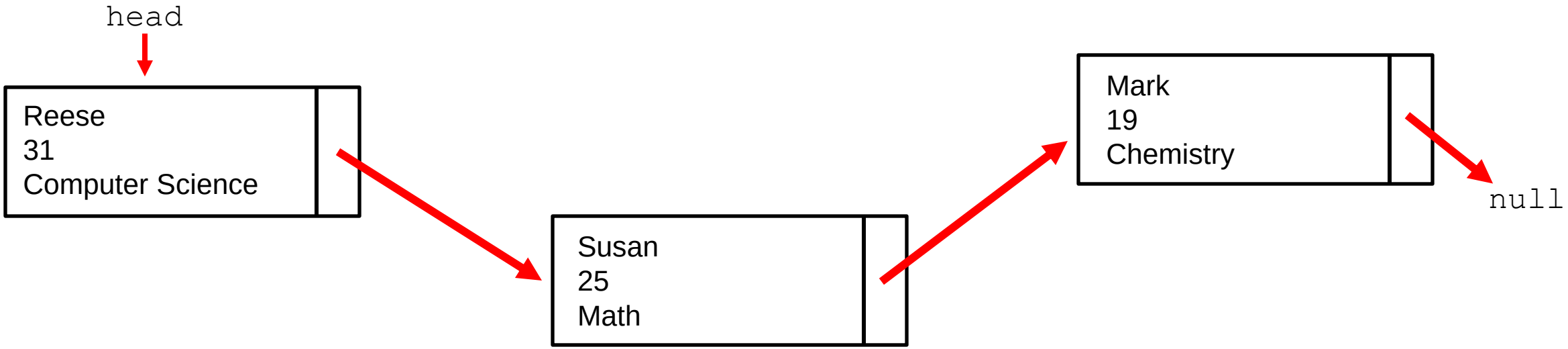
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Nodes consists of two parts:

1. Payload

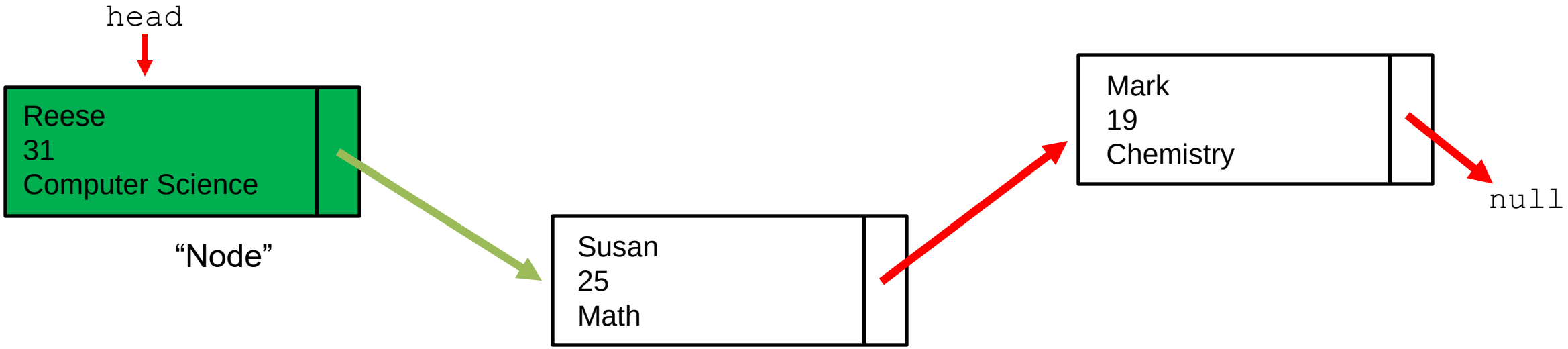
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Nodes consists of two parts:

1. Payload
2. Pointer to next node

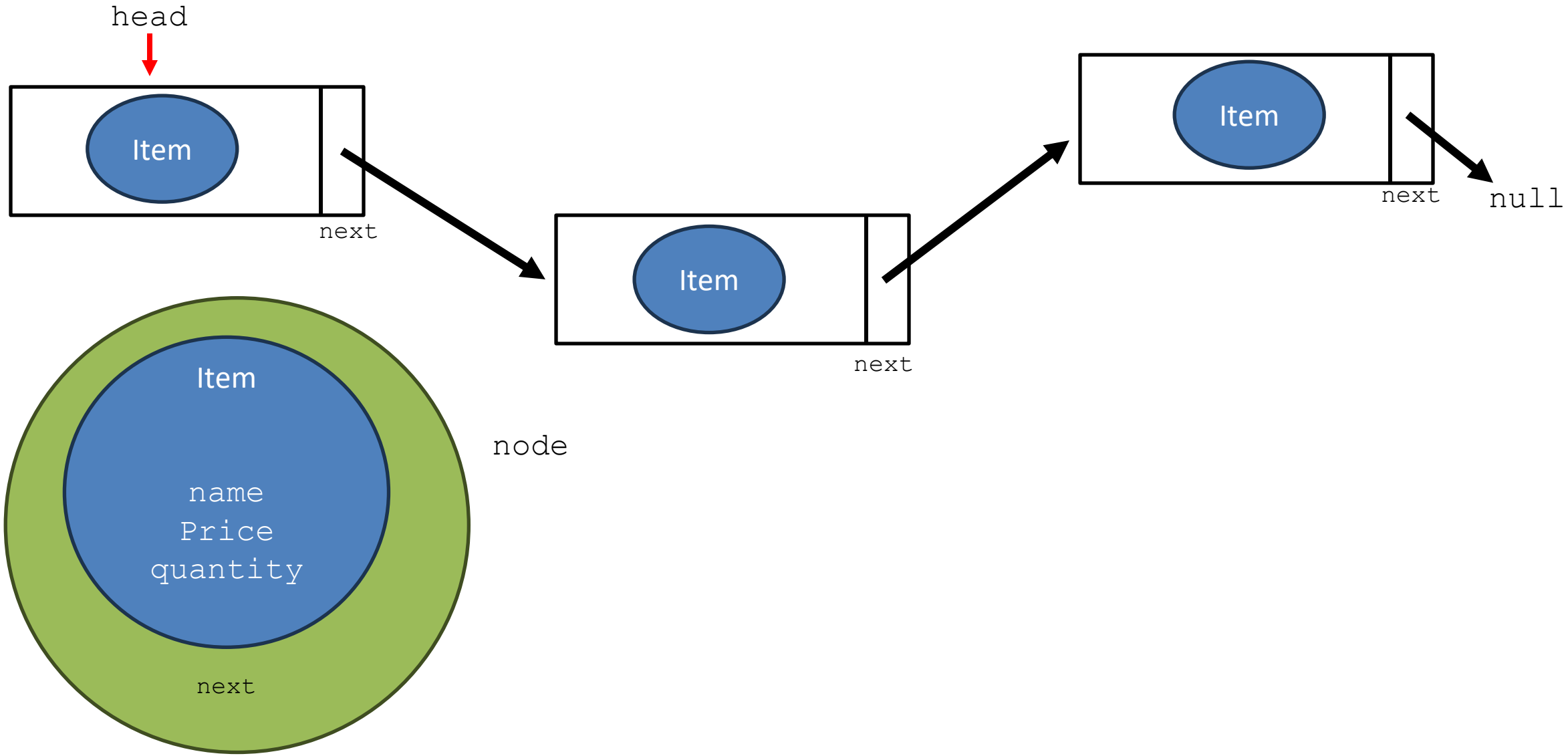
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



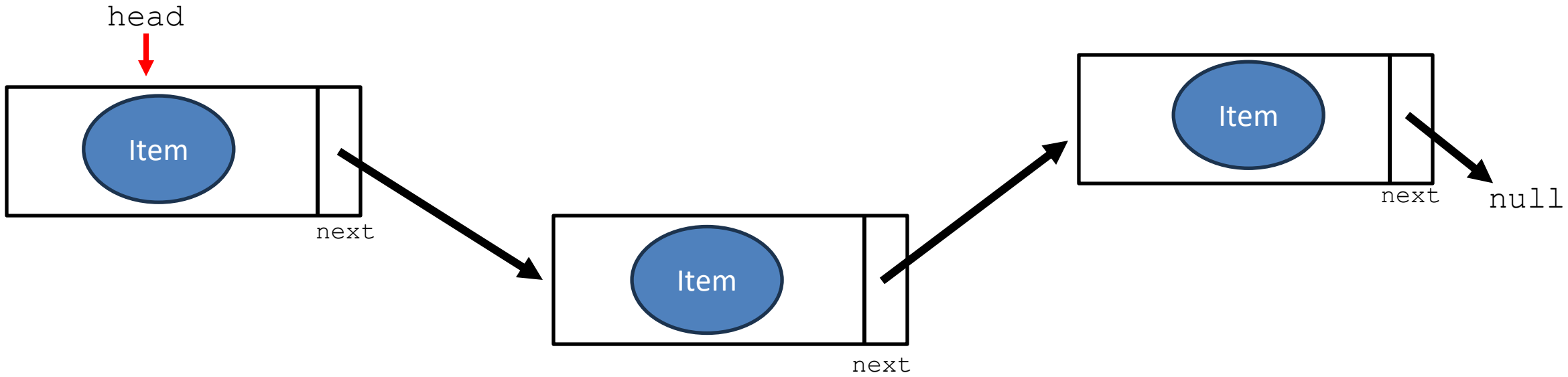
Nodes consists of two parts:

1. Payload
2. Pointer to next node

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)

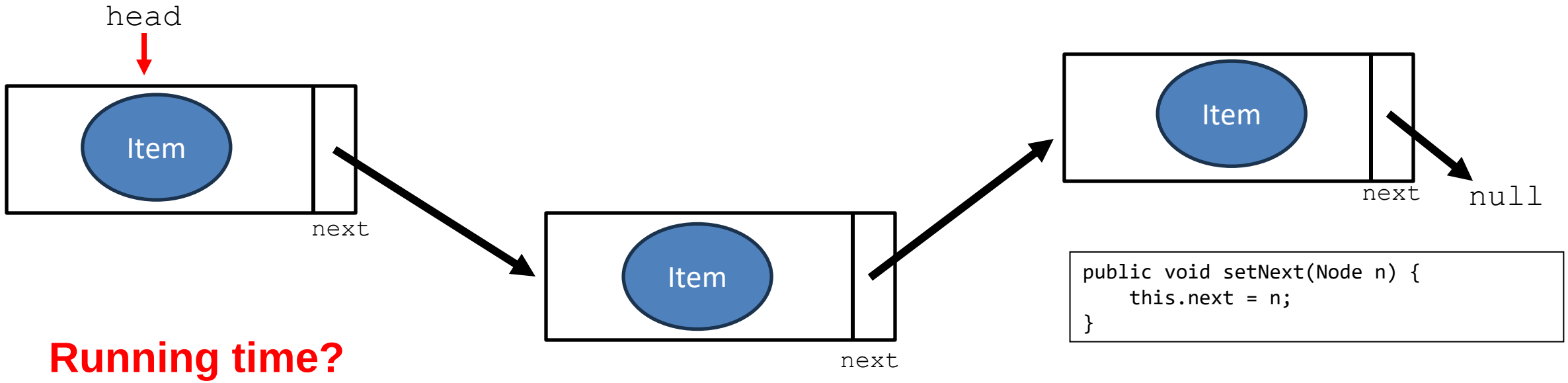


A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)




```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)

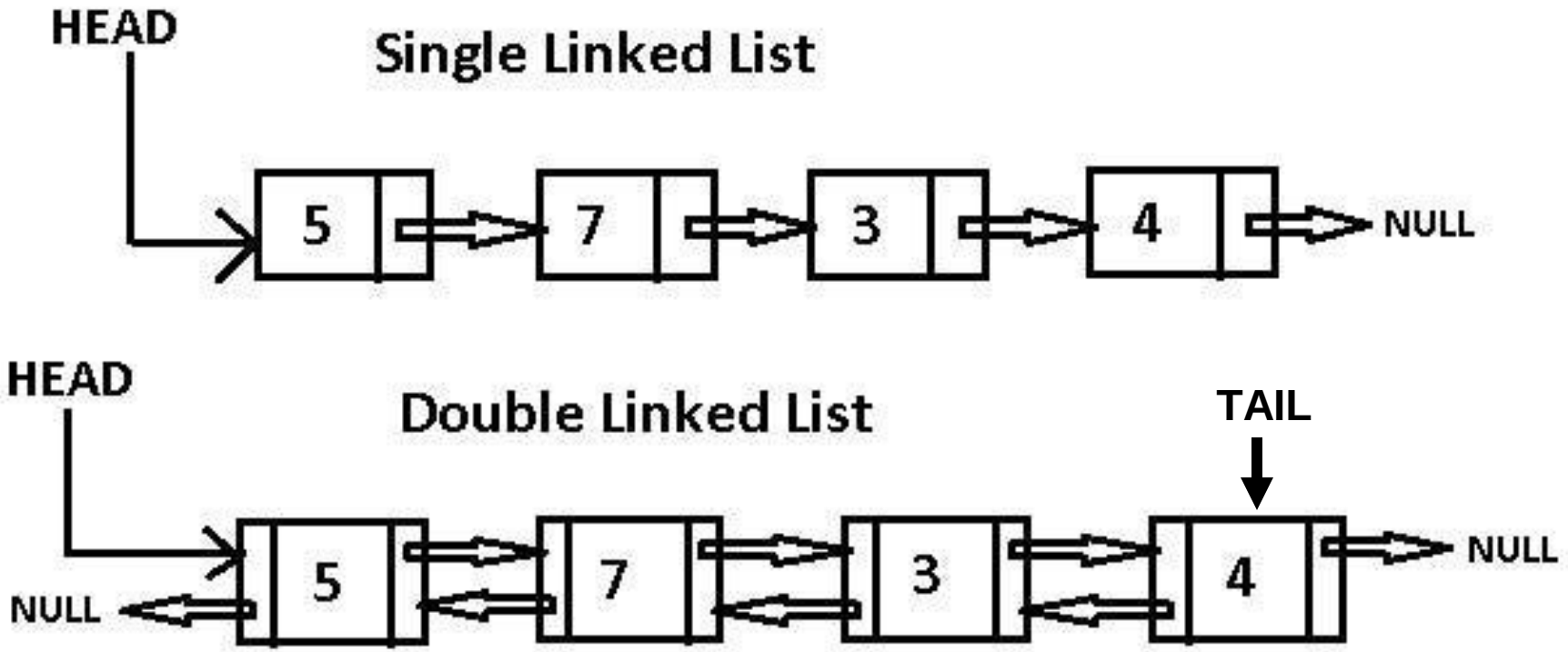



```
public void setNext(Node n) {
    this.next = n;
}
```

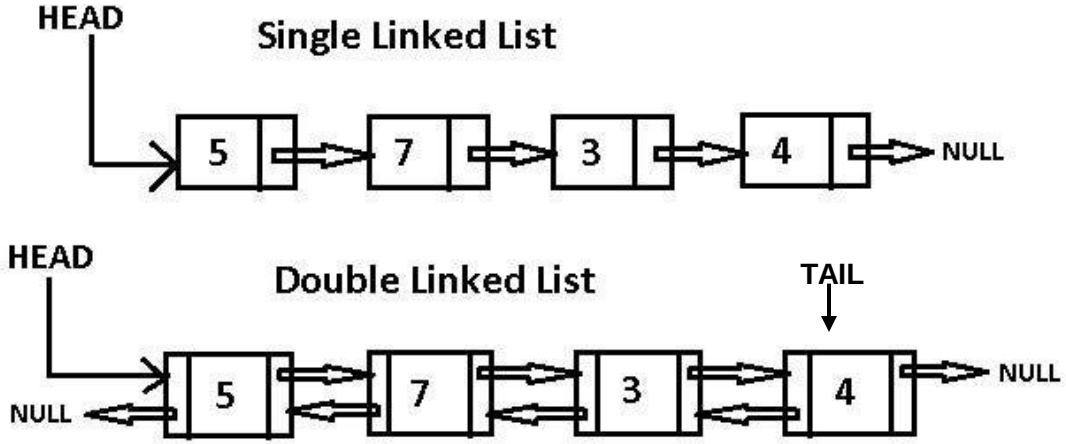
Running time?

```
public void addToFront(Node newNode) {
    if(head == null) {          O(1)
        head = newNode;         O(1)
    }
    else {
        newNode.setNext(head);  O(1)
        head = newNode;         O(1)
    }
}
```

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)

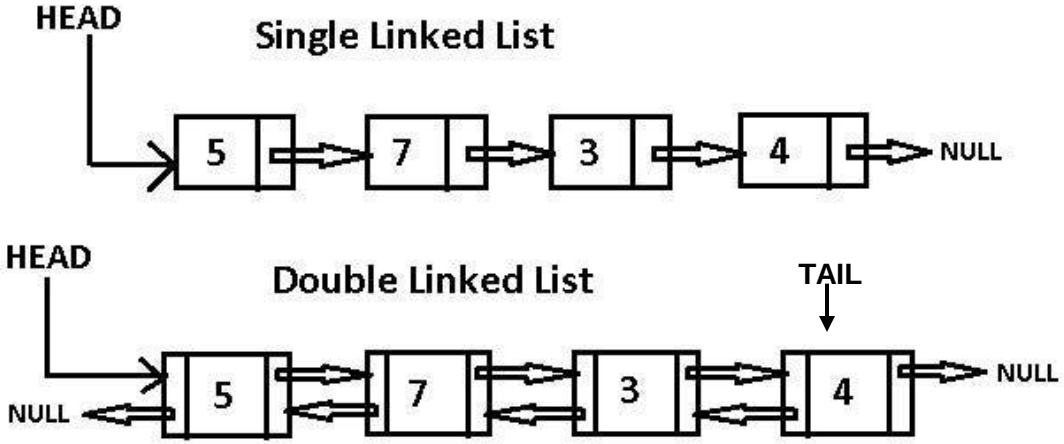


A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Operation	Time Complexity
Delete / Add first node	O(1)
Delete / Add tail node	O(1)
General Add/Delete node	O(n)
Linear Search	O(n)
Forward Traversal	O(n)

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Linked Lists

- Do not have indices
- Less memory efficient compared to arrays

Takeaway: Adding/Deleting to LL is O(1) work

(if adding to front or back)

Operation	Time Complexity
Delete / Add first node	O(1)
Delete / Add tail node	O(1)
General Add/Delete node	O(n)
Linear Search	O(n)
Forward Traversal	O(n)

A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)

We will never write our own Linked List class, instead we will always import the Linked List Java Library!

```
import java.util.LinkedList;
```

```java
import java.util.LinkedList;

public class march20demo {

    public static void main(String[] args) {

        LinkedList<String> names = new LinkedList<String>();

        names.add("Reese");
        names.add("Spencer");
        names.add("Susan");

        System.out.println(names);


    }

}
```

Method Summary

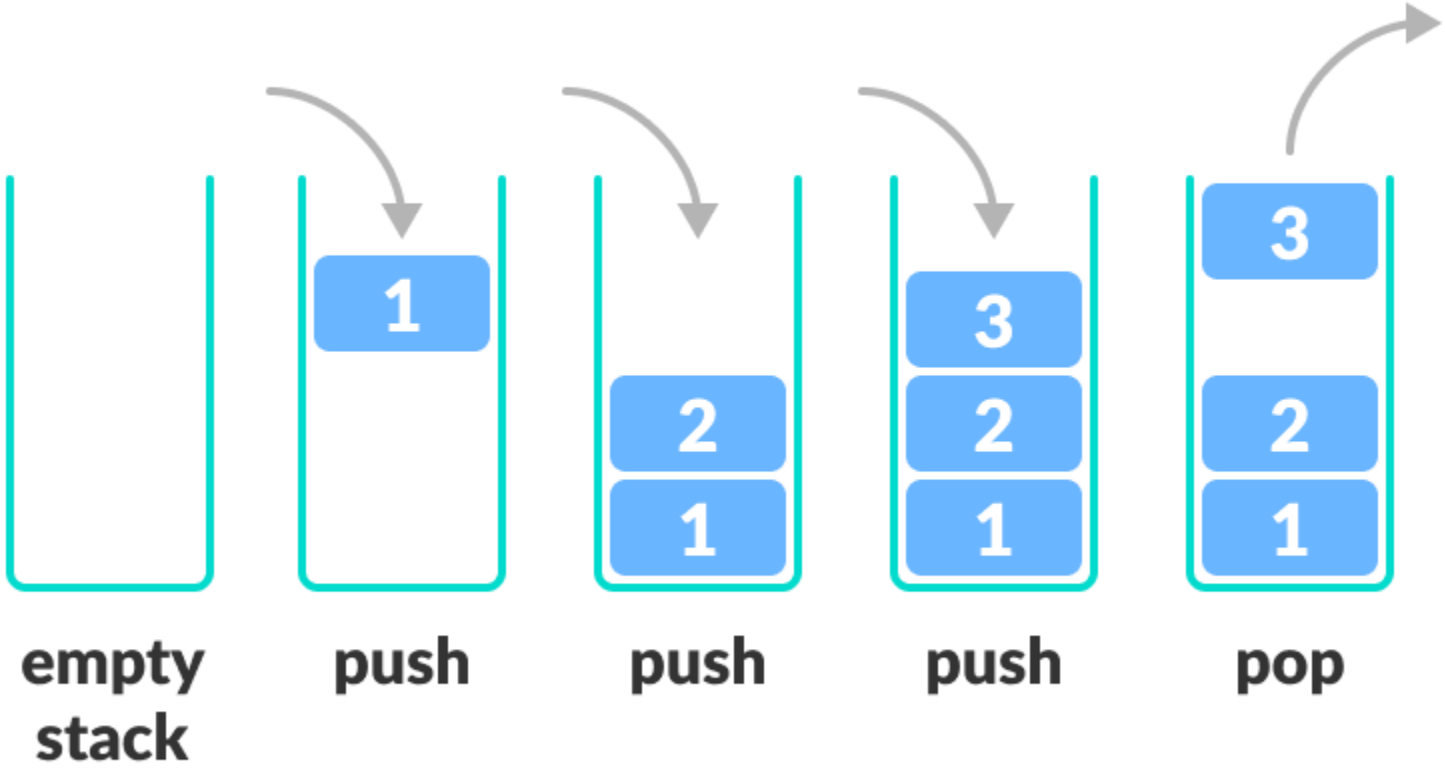
Methods

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	`addFirst(E e)` Inserts the specified element at the beginning of this list.
void	`addLast(E e)` Appends the specified element to the end of this list.
void	`clear()` Removes all of the elements from this list.
Object	`clone()` Returns a shallow copy of this `LinkedList`.
boolean	`contains(Object o)` Returns `true` if this list contains the specified element.
Iterator<E>	`descendingIterator()` Returns an iterator over the elements in this deque in reverse sequential order.
E	`element()` Retrieves, but does not remove, the head (first element) of this list.
E	`get(int index)` Returns the element at the specified position in this list.

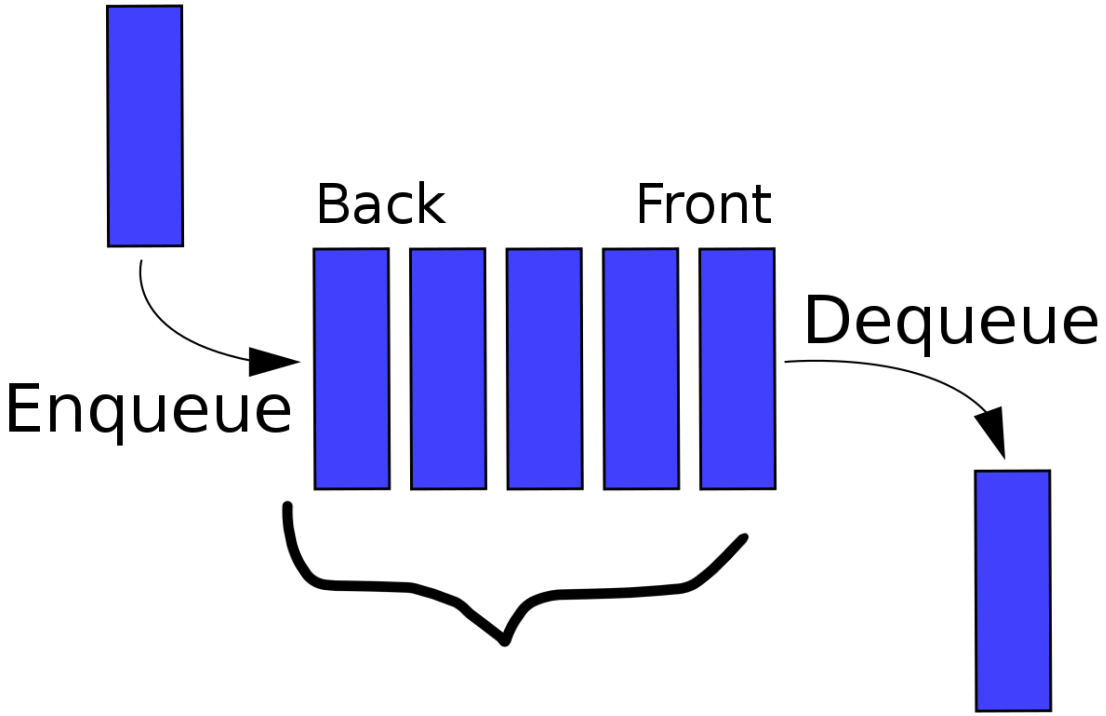
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

We can:

- Add an element to the top of the stack (push)
- Remove the top element (pop)

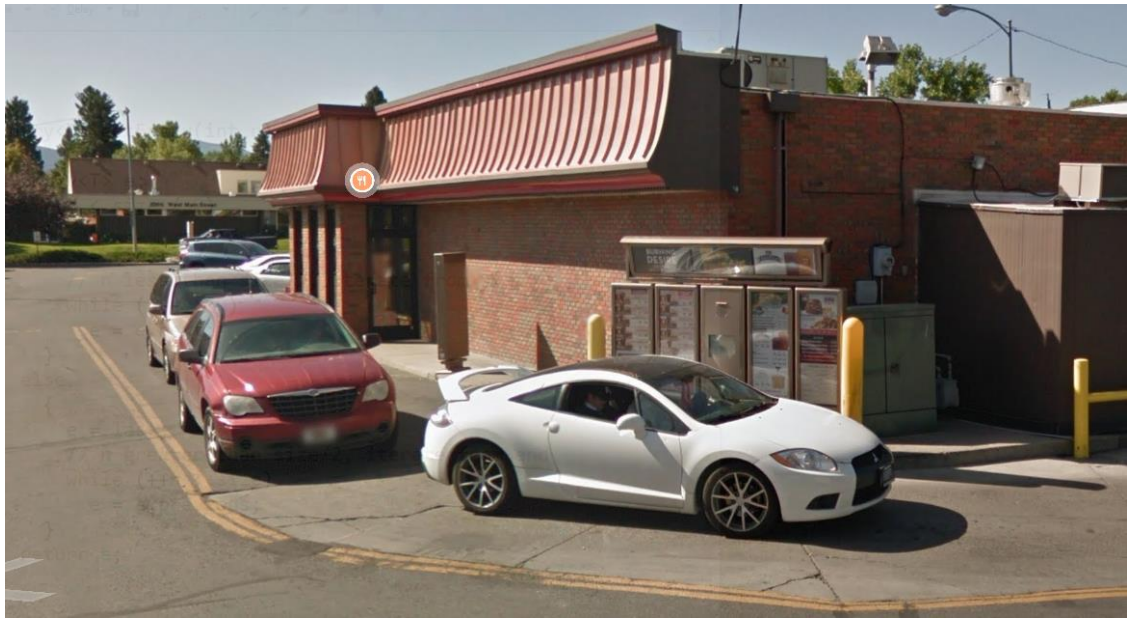


A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Elements get added to the **`Back`** of the Queue.

Elements get removed from the **`Front`** of the queue



Applications of Queue Data Structures

- Online waiting rooms
- Operating System task scheduling
- Web Server Request Handlers
- Network Communication
- CSCI 232 Algorithms

Queue Runtime Analysis

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Takeaway: Adding to stack or queue is O(1) work

Applications of Stack Data Structures

- Tracking function calls in programming
- Web browser history
- Undo/Redo buttons
- Recursion/Backtracking
- CSCI 232 Algorithms

Stack Runtime Analysis

	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

In CSCI 232, if we ever need to use a stack or queue, we will import the Java library!

```
import.java.util.Stack
```

```
import.java.util.Queue
```

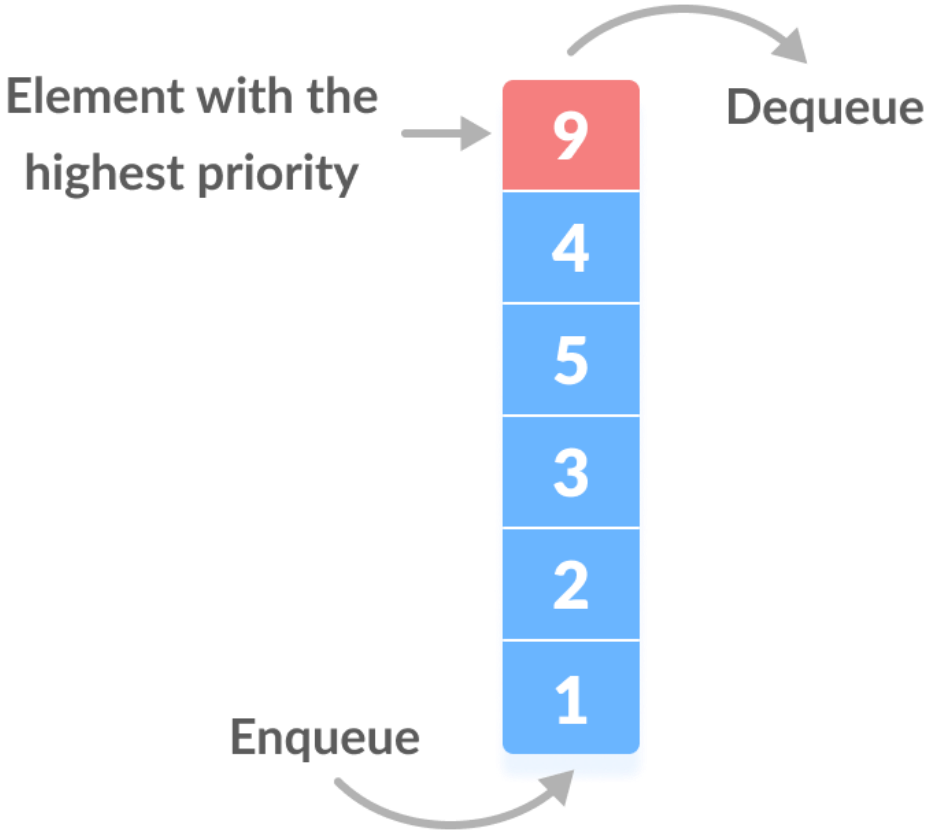
`java.util.Queue` is an interface. We cannot create a Queue object.
Instead, we create an instance of an object *that implements* this interface

Some of the Classes that implement the Queue interface:

1. PriorityQueue (`java.util.PriorityQueue`)
2. Linked List (`java.util.LinkedList`)

(If you need a FIFO queue, Linked List is the way to go…)

Most of the time, queues will operate in a FIFO fashion, however there may be times we want to dequeue the item with the **highest priority**



Priority queue in a data structure is an extension of a linear queue that possesses the following properties: Every element has a certain priority assigned to it

When we enqueue something, we might need to “shuffle” that item into the correct spot of the priority queue

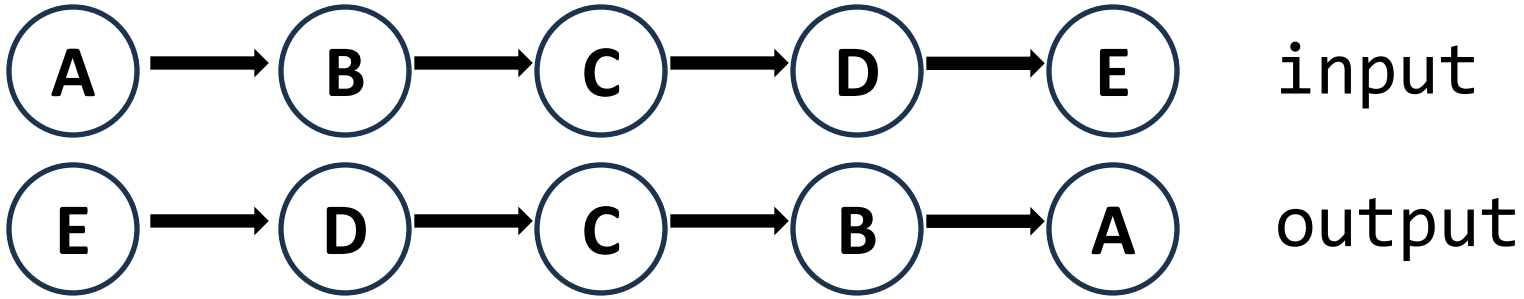
Sorting

Bubble Sort	O(n^2)
Selection Sort	O(n^2)
Merge Sort	O(nlogn)
Quick Sort	O(nlogn) (on average)

Takeaway: the fastest sorting algorithm known (currently) is O(nlogn)

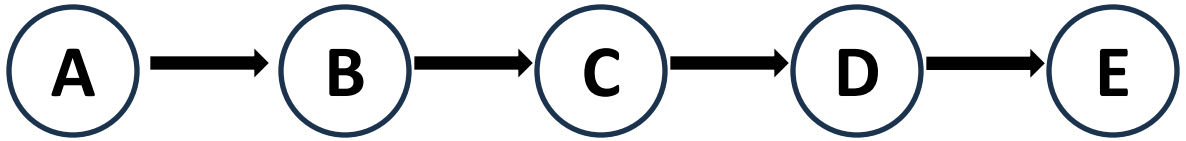
(Why don't you think there are any O(1) or O(logn) sorting algorithms?)

LeetCode #206 Reverse a (singly) Linked List




```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {

    }
}
```

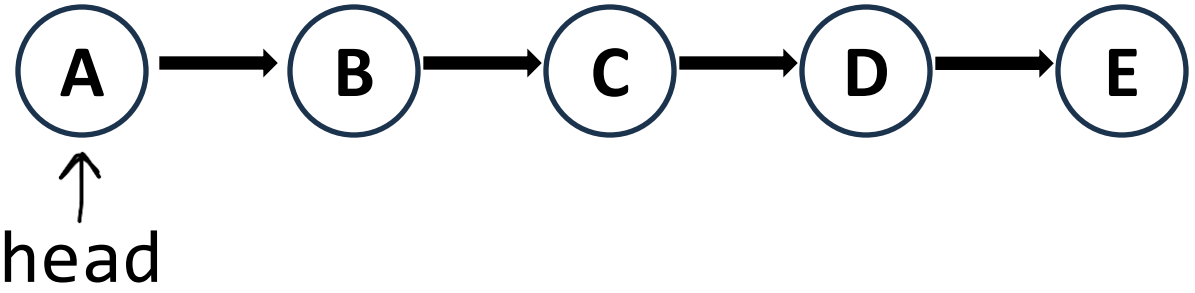
LeetCode #206 Reverse a (singly) Linked List



input

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){


  }

  return ???;
}
```

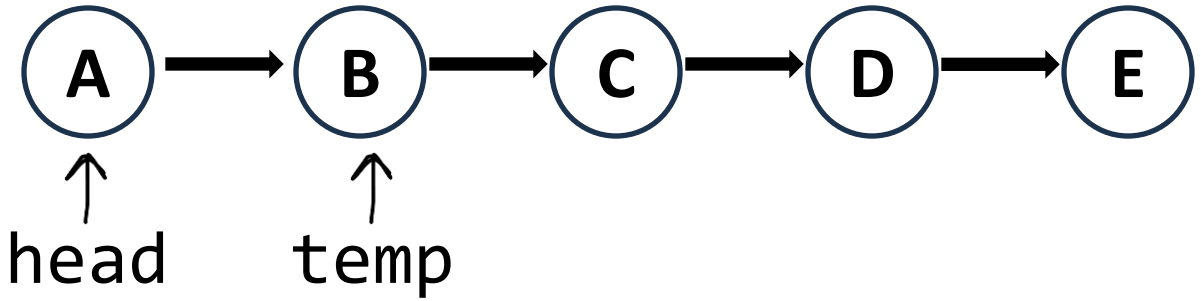
LeetCode #206 Reverse a (singly) Linked List



prev → null

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){


  }

  return ???;
}
```

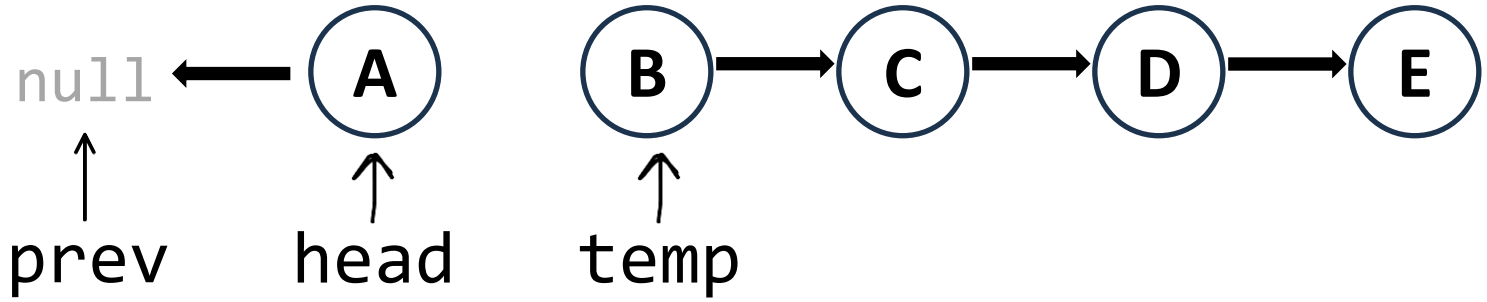
LeetCode #206 Reverse a (singly) Linked List



prev → null

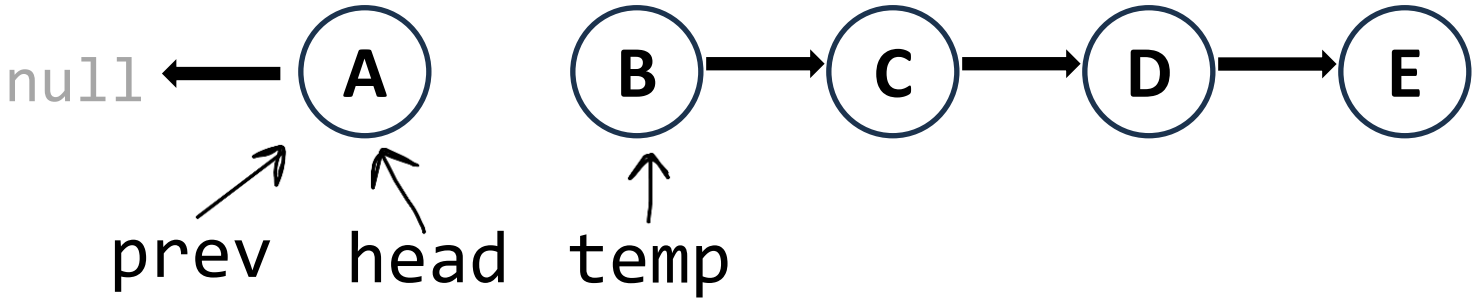
```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;



  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



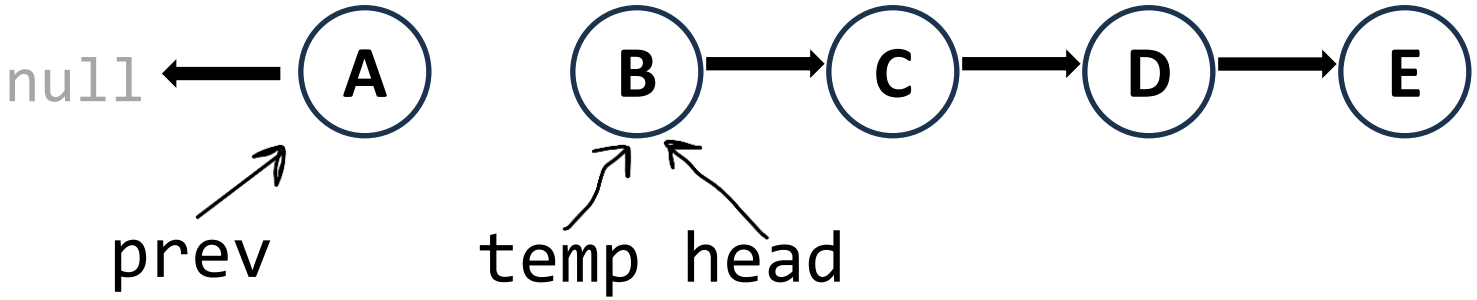

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;



  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



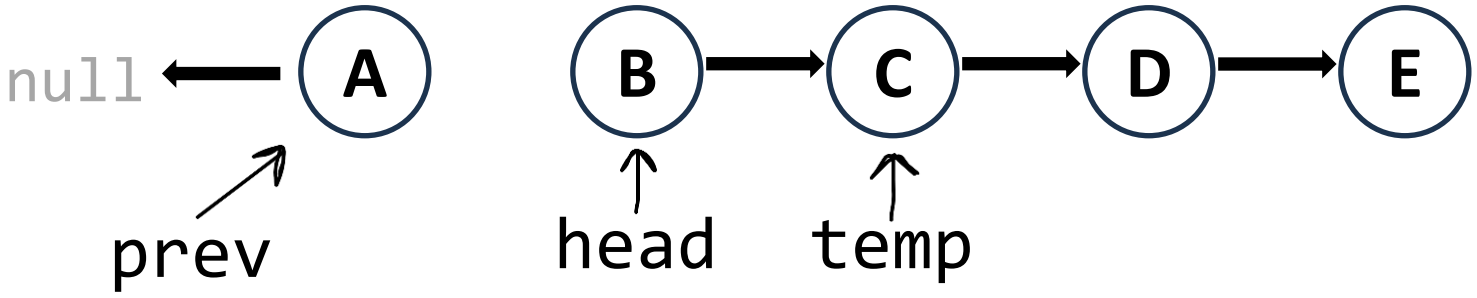

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;

  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



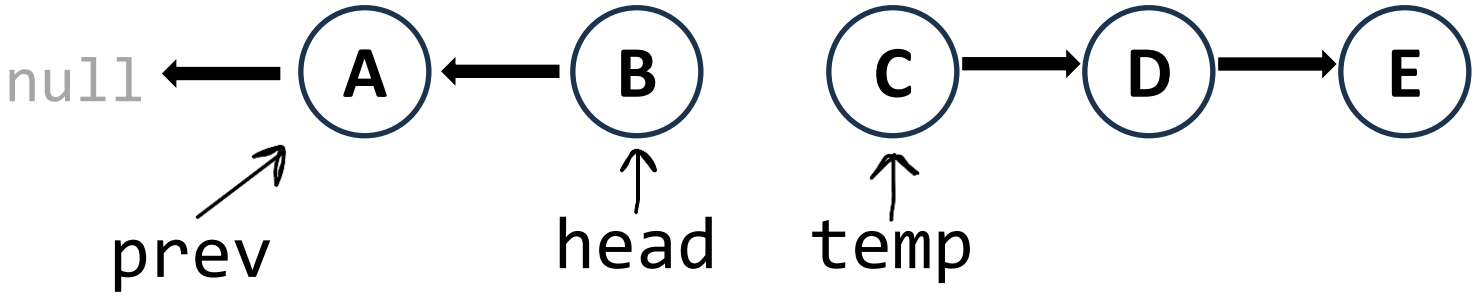

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



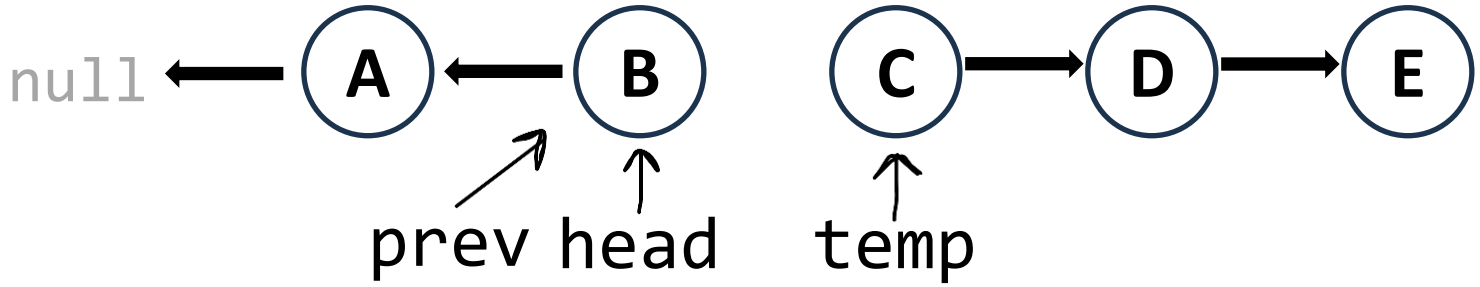

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



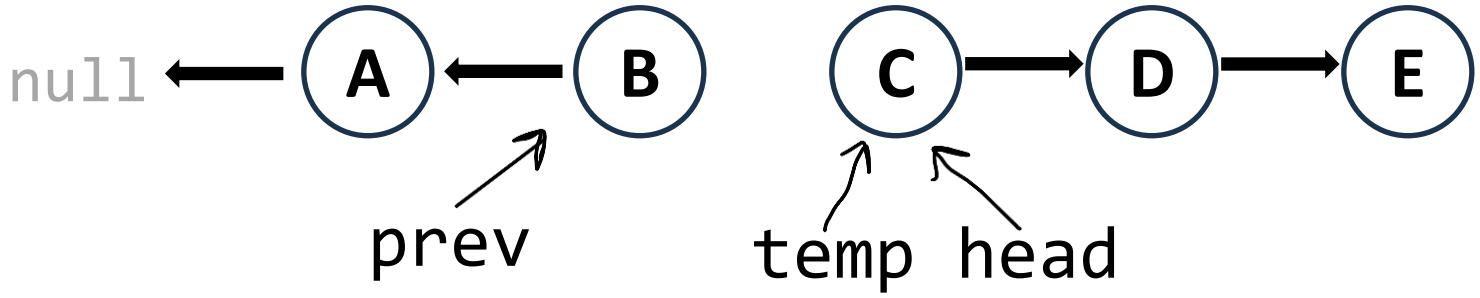

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



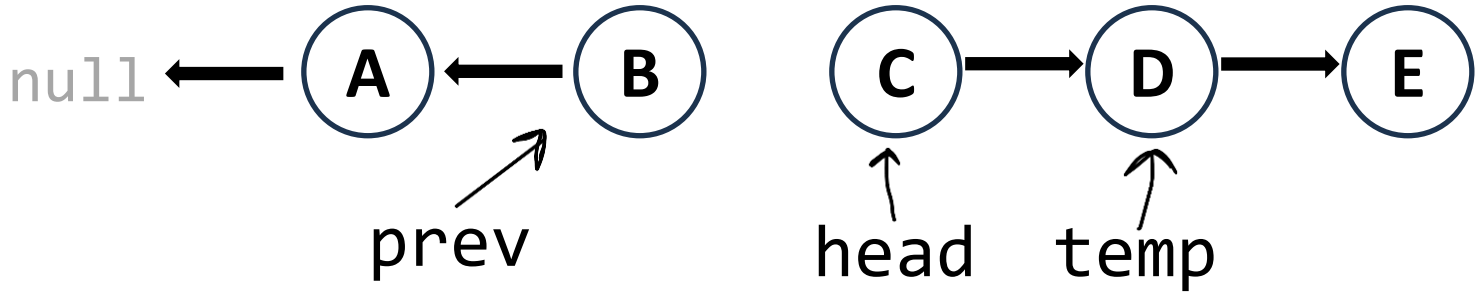

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



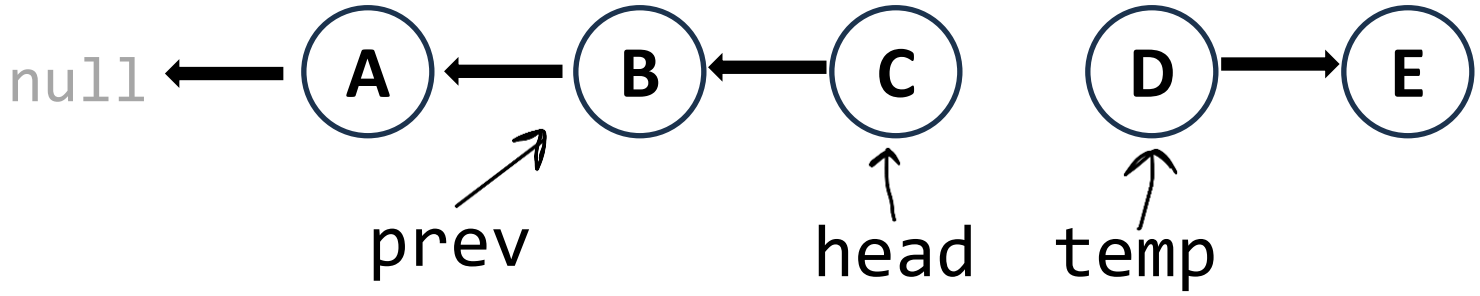

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



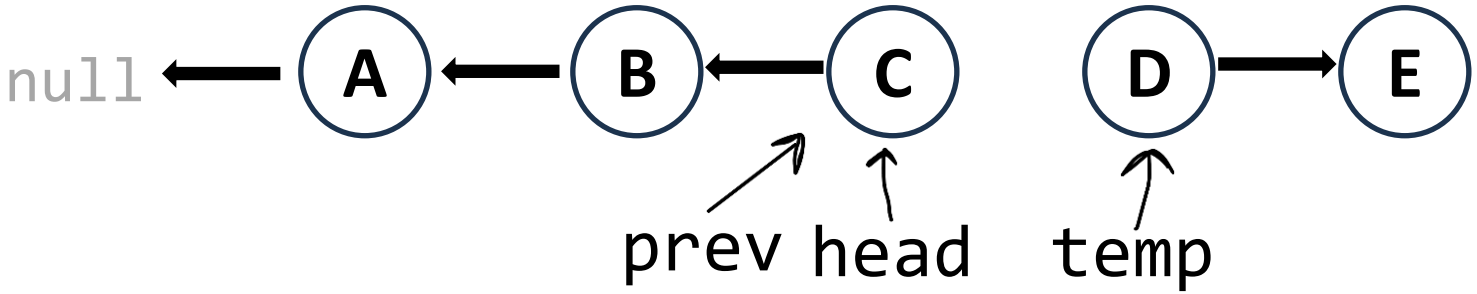

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



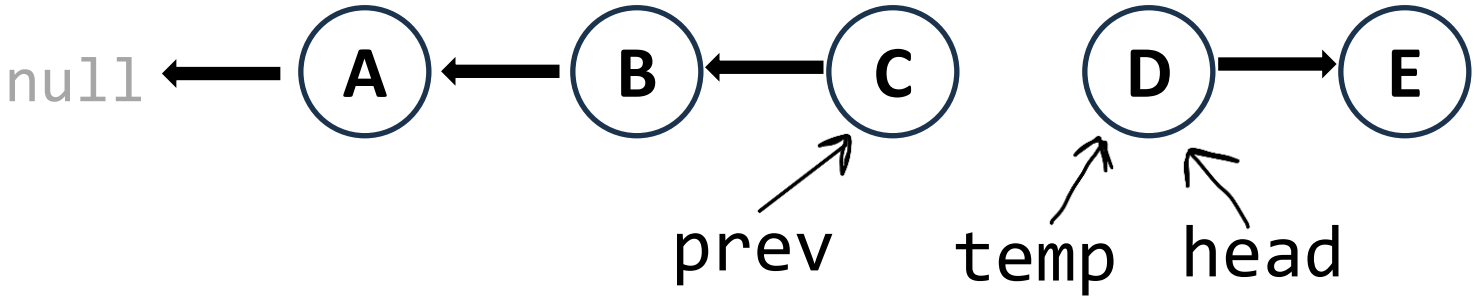

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



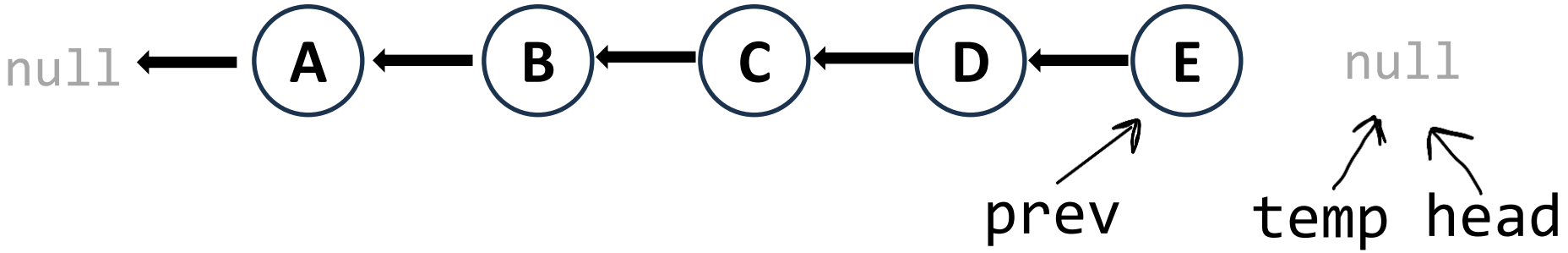

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



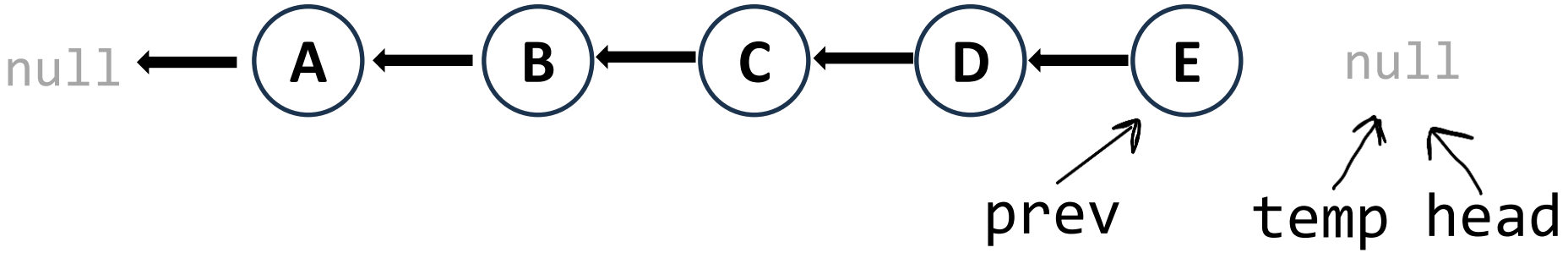

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List



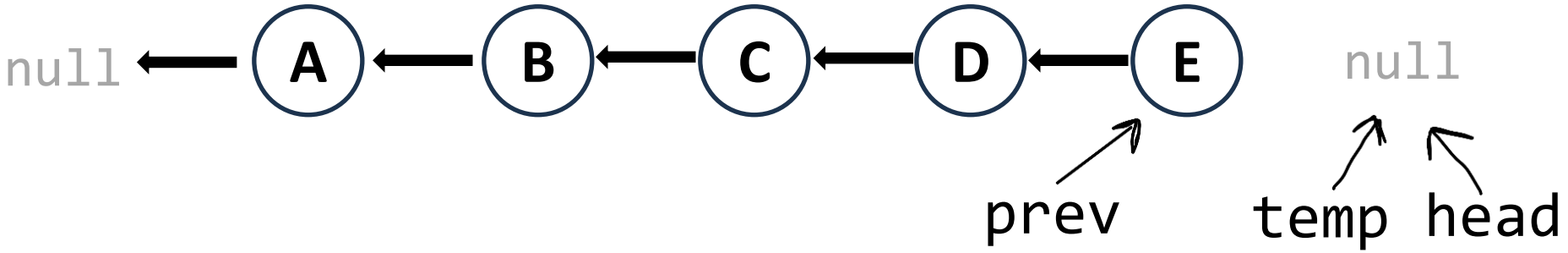

```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return ???;
}
```

LeetCode #206 Reverse a (singly) Linked List

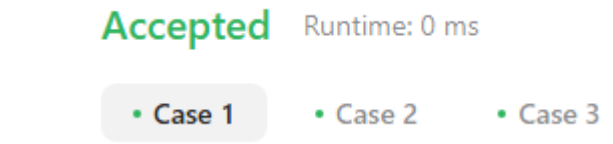



```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return prev;
}
```

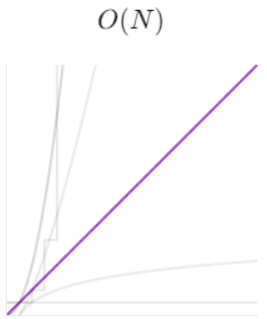
LeetCode #206 Reverse a (singly) Linked List




```
public ListNode reverseList(ListNode head){

  ListNode prev = null;

  while(head != null){
       ListNode temp = head.next;
       head.next = prev;
       prev = head;
       head = temp;
  }

  return prev;
}
```



Time Complexity



Lab 1


LinkedList
LinkedListElement
LinkedListElement
LinkedListElement
LinkedListElement
LinkedListElement