CSCI 232:
Data Structures and Algorithms

Java Review

Reese Pearsall

Summer 2025

We are going to write a program where a user can keep track of their online shopping cart.

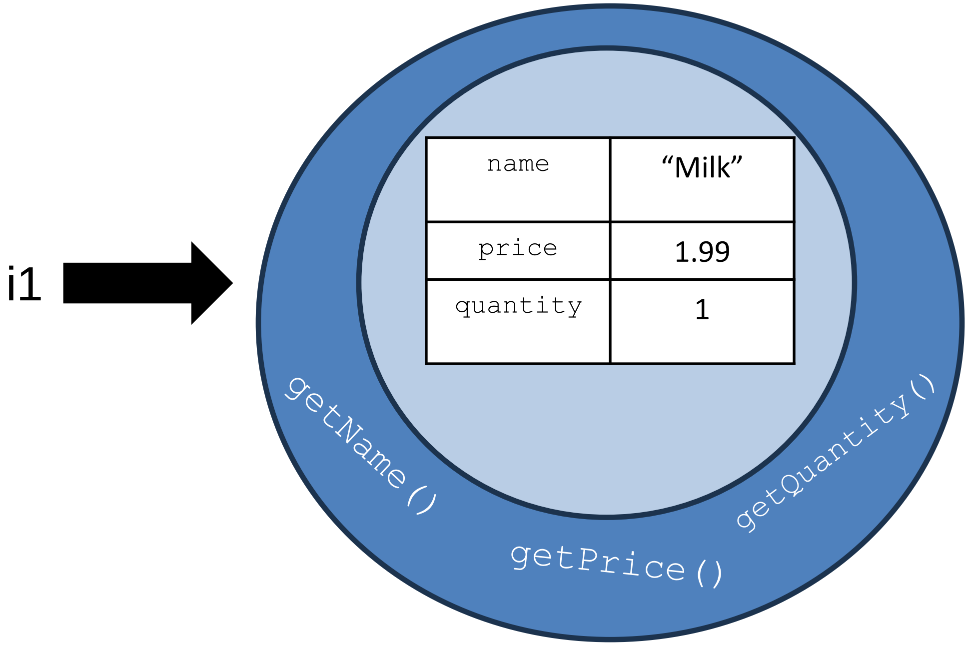
Users can add items, remove items, search for items, get the total price of cart, and apply coupons to items

```java
public class Item {


    private String name;
    private double price;
    private int quantity;

    public Item(String n, double p, int q) {
        this.name = n;
        this.price = p;
        this.quantity = q;
    }

    public String getName() {
        return this.name;
    }

    public double getPrice() {
        return this.price;
    }

    public int getQuantity() {
        return this.quantity;
    }
}
```

Java Class: Blueprint for an object (i.e. a "thing")

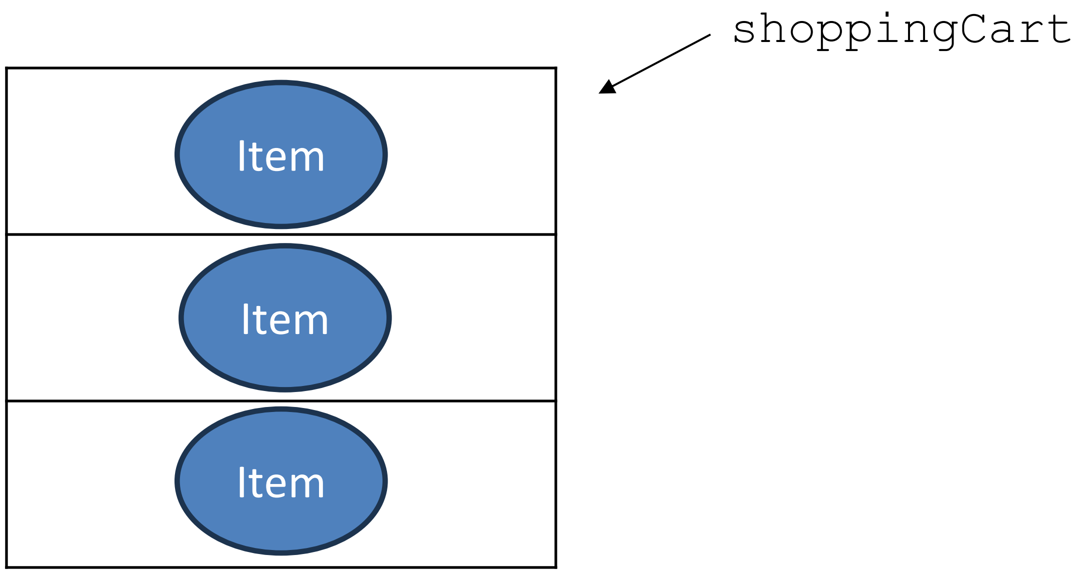
- Instance Field/Attributes
- Methods

```java
Item i1 = new Item("Milk", 1.99, 1);
Item i2 = new Item("Eggs", 3.99, 2);

System.out.println(i1.getName());
System.out.println(i2.getQuantity());
```

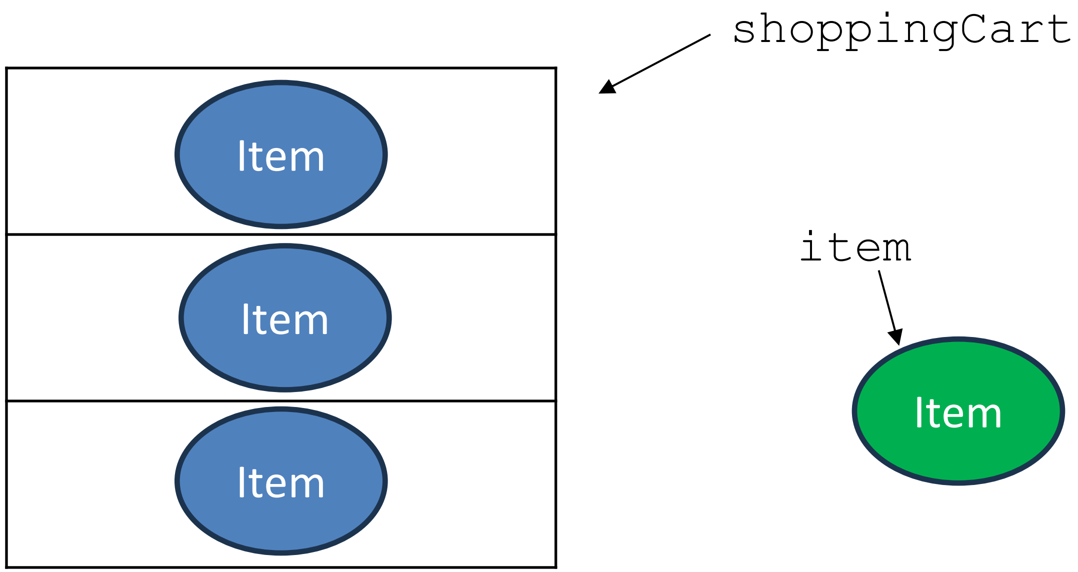


Java Objects: **Instances** of classes.
Program entities

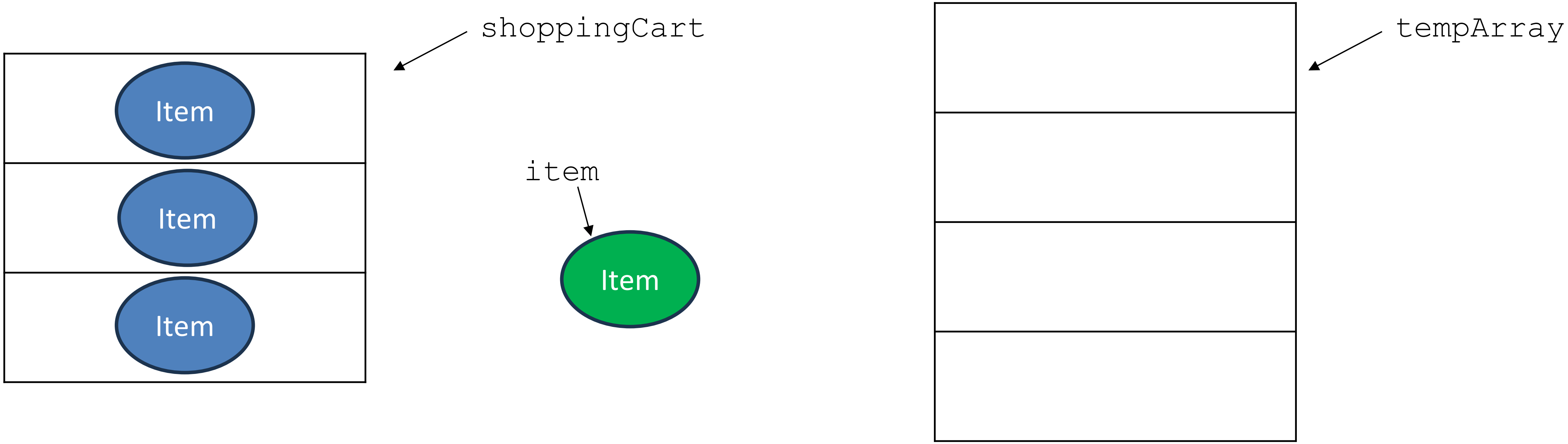
```java
public void addItem(String name, double price, int quantity) {
	Item item = new Item(name, price, quantity);
	Item[] tempArray = new Item[this.shoppingCart.length + 1];
	for(int i = 0; i < this.shoppingCart.length; i++) {
		tempArray[i] = shoppingCart[i];
	}
	tempArray[shoppingCart.length] = item;
	shoppingCart = tempArray;
	this.num_of_items++;
}
```



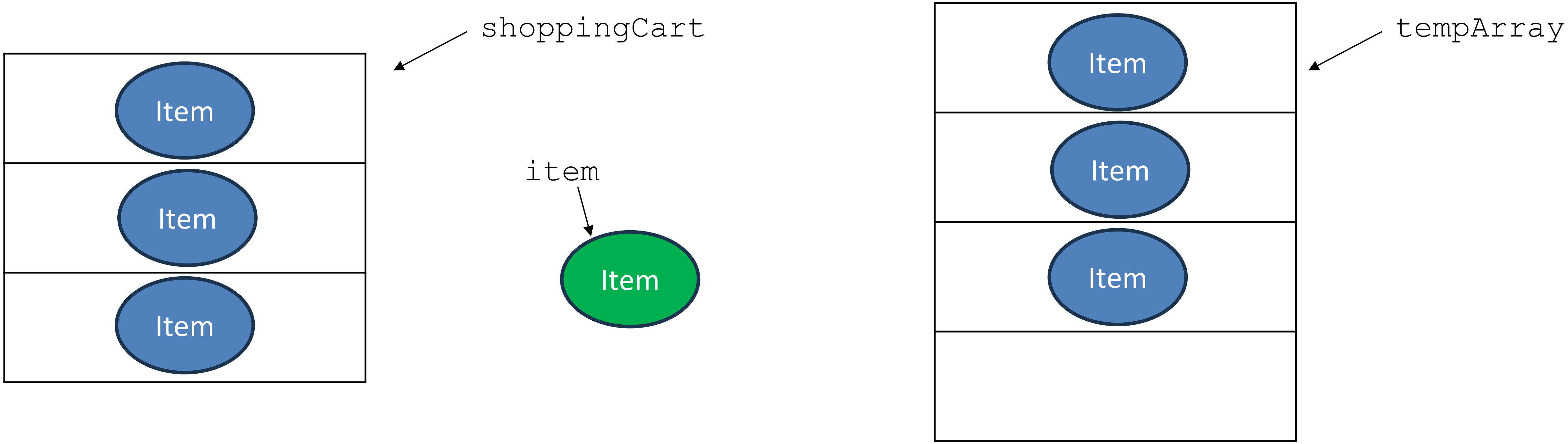
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



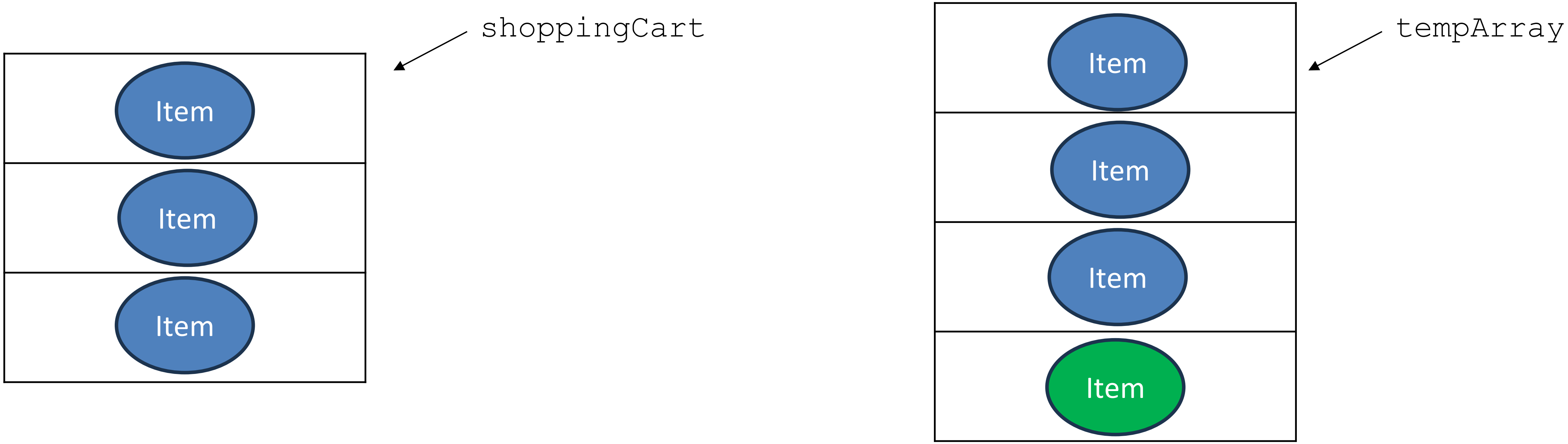
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



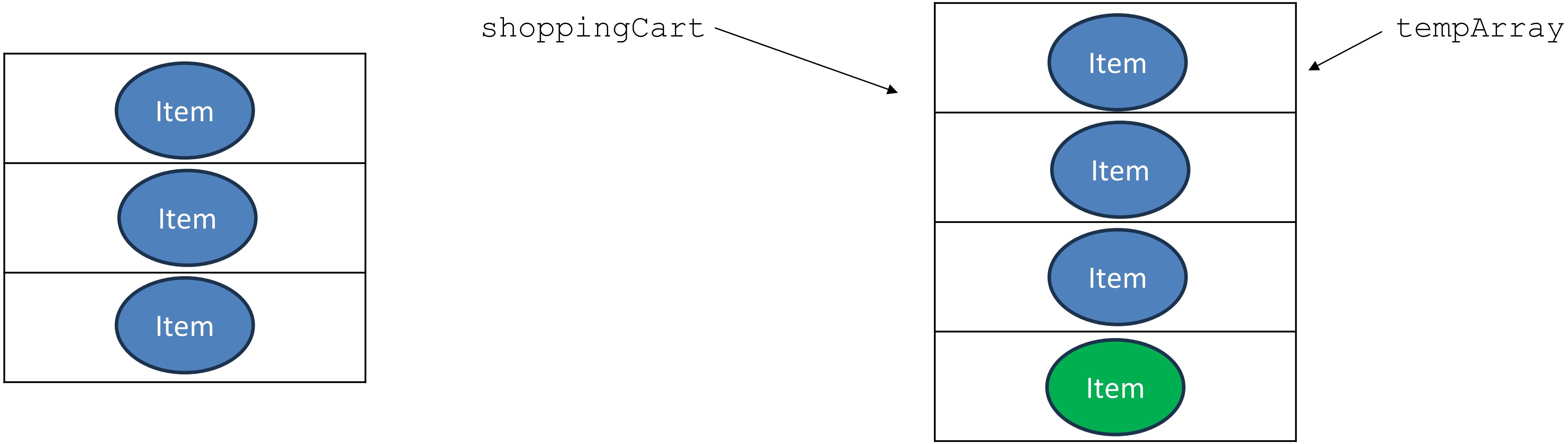
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time?

```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

Big O Notation: Upper bound on asymptotic growth. I.e. Worst case upper bound of a function

```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

Big O Notation: Upper bound on asymptotic growth. I.e. Worst case upper bound of a function

Big O Notation measures the number of steps needed to complete an algorithm under the worst-case scenario

```java
public int linearSearch(int[] array, int target) {
        for(int i = 0; i < array.length; i++) {
                if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

```
public int linearSearch(int[] array, int target) {
 ??? → for(int i = 0; i < array.length; i++) {
                if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Worst case scenario, this for loop will need run **n** times

O(n) **Let n = array.length**

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(???) → if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
       O(???) → if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
         O(1) → if(array[i] == target){
                         return i;
                    }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(1) → if(array[i] == target){
                O(1) → return i;
                }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(1) → if(array[i] == target){
                O(1) → return i;
                }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

Total running time: O(n * 1 + 1)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
          O(1) → if(array[i] == target){
                   O(1) → return i;
                 }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

Total running time: O(n * 1 + 1)

In Big O notation:

- We can drop non dominant factors
- We can drop multiplicative constants (coefficients)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
          O(1) → if(array[i] == target){
                   O(1) → return i;
                 }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

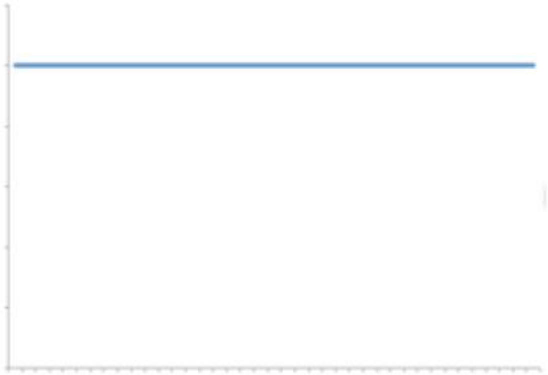
Primitive operation – operation that takes constant time (independent of size of the input)

Total running time: O(n) where n = | array |

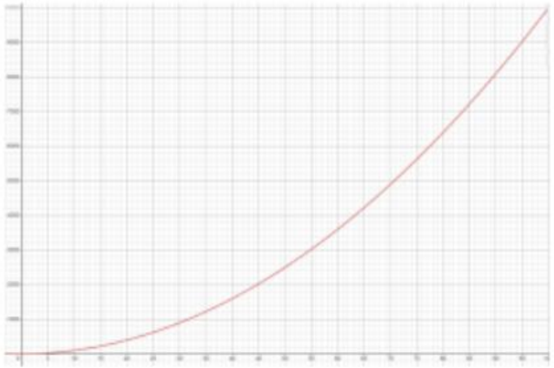
In Big O notation:

- We can drop non dominant factors
- We can drop multiplicative constants (coefficients)

Constant



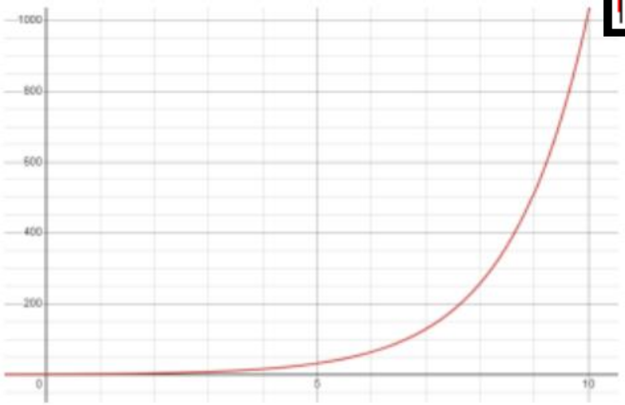
Quadratic

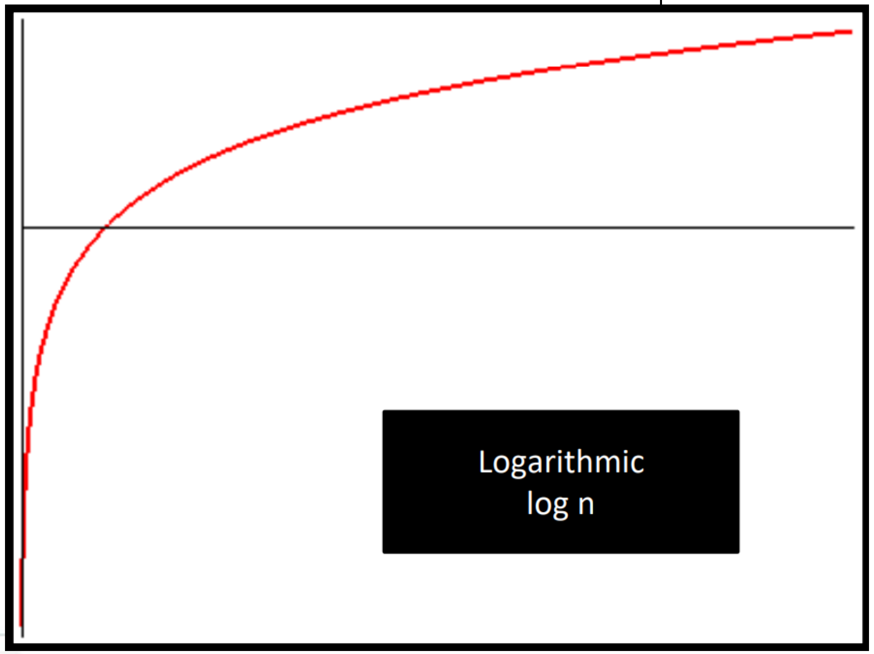


```
{
+ 1];
```

Linear

Exponential





```
function computeDistanceBetweenCaves():

	for each cave in all_caves i;
		for each cave in all_caves j;
			compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

```
function computeDistanceBetweenCaves():

  O(n)  for each cave in all_caves i;
        O(n-1) for each cave in all_caves j;
               O(1) compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

```
function computeDistanceBetweenCaves():

O(n) for each cave in all_caves i;
        O(n) for each cave in all_caves j;
                O(1) compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

```
function computeDistanceBetweenCaves():

O(n) for each cave in all_caves i;
       O(n) for each cave in all_caves j;
              O(1) compute_distance(i, j)
```

	H1	H2	H3	…	H9
H1	/	D(1,2)	D(1,3)	…	D(1,9)
H2	D(2,1)	/	D(2,3)	…	D(2,9)
H3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
H9	D(9,1)	D(9,2)	D(9,3)	…	/

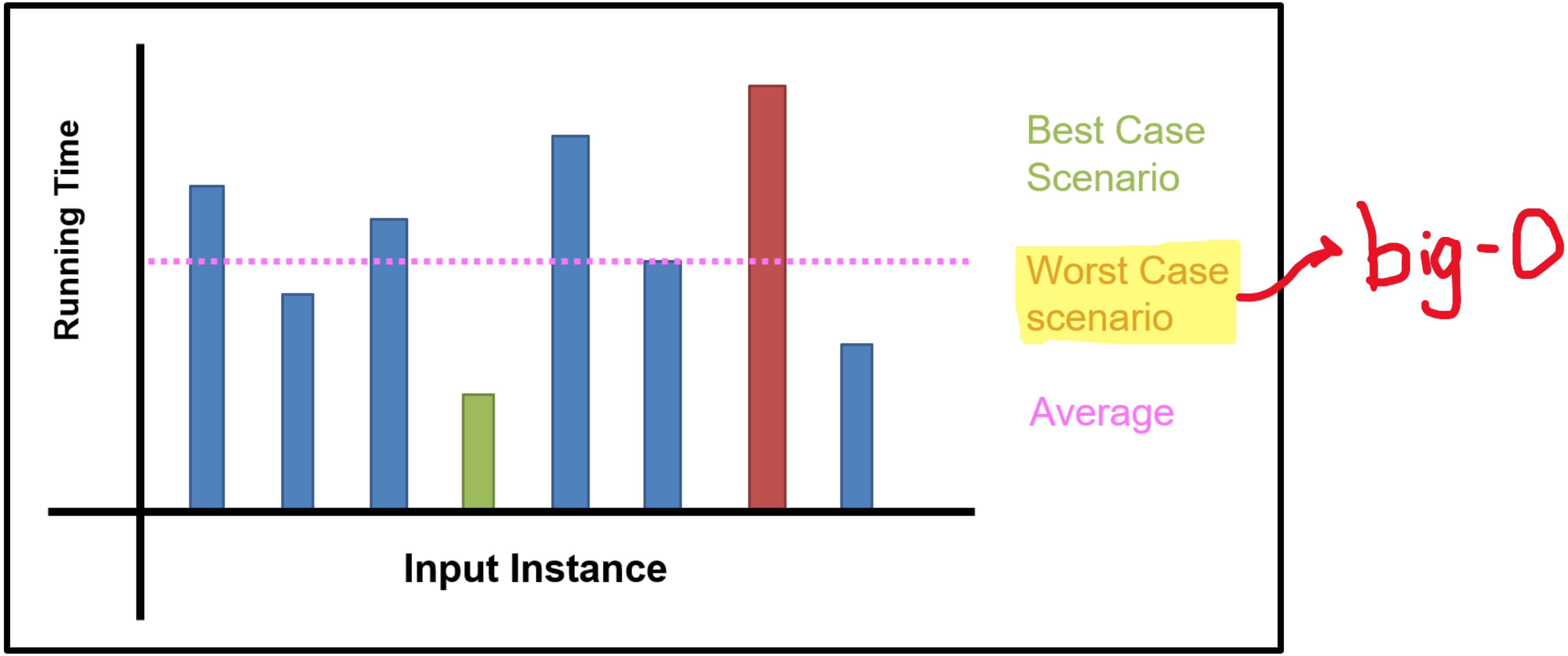
Total running time = O(n) * (O(n) * O(1))

```
function computeDistanceBetweenCaves():

O(n)  for each cave in all_caves i;
      O(n)  for each cave in all_caves j;
            O(1) compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

Total running time = O(n) * (O(n) * O(1))

O(n^2) **Where n = # of caves**



In computer science (and this class in particular), we will be focusing on stating running time in terms of **worst-case scenario**

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant

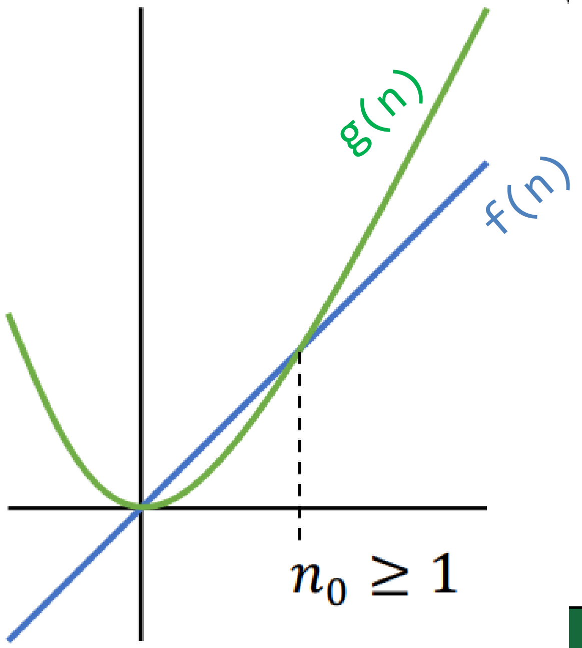
Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant



$$\forall n \geq 1, n^2 \geq n$$
$$\Rightarrow n \in O(n^2)$$

O -notation provides an upper bound on some function $f(n)$

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $O(n^2)$ time.

Algorithm **B** runs in $O(n)$ time.

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

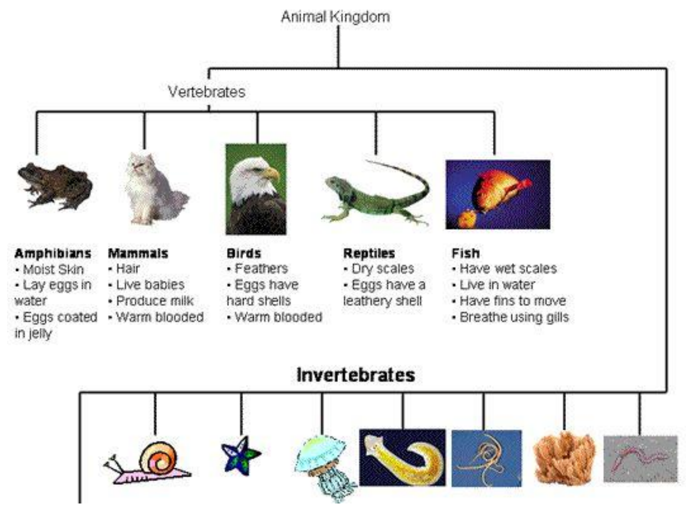
Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

Big-O is a helpful way to broadly describe the running time of different programs, but it isn't perfect



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

```java
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
   O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
   O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
         }
         tempArray[shoppingCart.length] = item;
         shoppingCart = tempArray;
         this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
          O(1) → tempArray[i] = shoppingCart[i];
         }
         tempArray[shoppingCart.length] = item;
         shoppingCart = tempArray;
         this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
             O(1) → tempArray[i] = shoppingCart[i];
         }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
          O(1) → tempArray[i] = shoppingCart[i];
         }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
 O(n) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
           O(1) → tempArray[i] = shoppingCart[i];
         }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

Total running time: O(n) + O(n)

O(2n)

O(n)   where n = shoppingCart.length

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
 O(n) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
 O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
          O(1) → tempArray[i] = shoppingCart[i];
        }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

Total running time: O(n) + O(n)

O(2n)

O(n) where n = shoppingCart.length

Takeaway: Adding to a full array takes O(n) time

```java
private static int binary_search(int[] array, int n) {
        int low = 0;
        int high = array.length - 1;
        while(low <= high) {
                int mid = (low + high) / 2;
                if(n == array[mid]) {
                        return mid;
                }
                else if(n > array[mid]) {
                        low = mid + 1;
                }
                else {
                        high = mid - 1;
                }
        }
        return -1;
}
```

Target Value: 27

1	**2**	**9**	**10**	**11**	**15**	**18**	**21**	**27**	**31**	**41**	**43**	**50**

```
private static int binary_search(int[] array, int n) {
	int low = 0;
	int high = array.length - 1;
	while(low <= high) {
		int mid = (low + high) / 2;
		if(n == array[mid]) {
			return mid;
		}
		else if(n > array[mid]) {
			low = mid + 1;
		}
		else {
			high = mid - 1;
		}
	}
	return -1;
}
```

Target Value: 27

1	2	9	10	11	15	18	21	27	31	41	43	50

```java
private static int binary_search(int[] array, int n) {
	int low = 0;
	int high = array.length - 1;
	while(low <= high) {
		int mid = (low + high) / 2;
		if(n == array[mid]) {
			return mid;
		}
		else if(n > array[mid]) {
			low = mid + 1;
		}
		else {
			high = mid - 1;
		}
	}
	return -1;
}
```

Target Value: 27

1	2	9	10	11	15	18	21	27	31	41	43	50

```java
private static int binary_search(int[] array, int n) {
	int low = 0;
	int high = array.length - 1;
	while(low <= high) {
		int mid = (low + high) / 2;
		if(n == array[mid]) {
			return mid;
		}
		else if(n > array[mid]) {
			low = mid + 1;
		}
		else {
			high = mid - 1;
		}
	}
	return -1;
}
```

Target Value: 27

1	2	9	10	11	15	18	21	27	31	41	43	50

```java
private static int binary_search(int[] array, int n) {
	int low = 0;
	int high = array.length - 1;
	while(low <= high) {
		int mid = (low + high) / 2;
		if(n == array[mid]) {
			return mid;
		}
		else if(n > array[mid]) {
			low = mid + 1;
		}
		else {
			high = mid - 1;
		}
	}
	return -1;
}
```

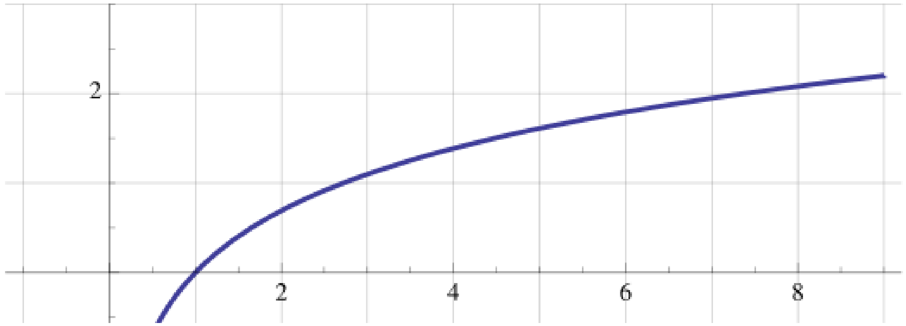
Target Value: 27

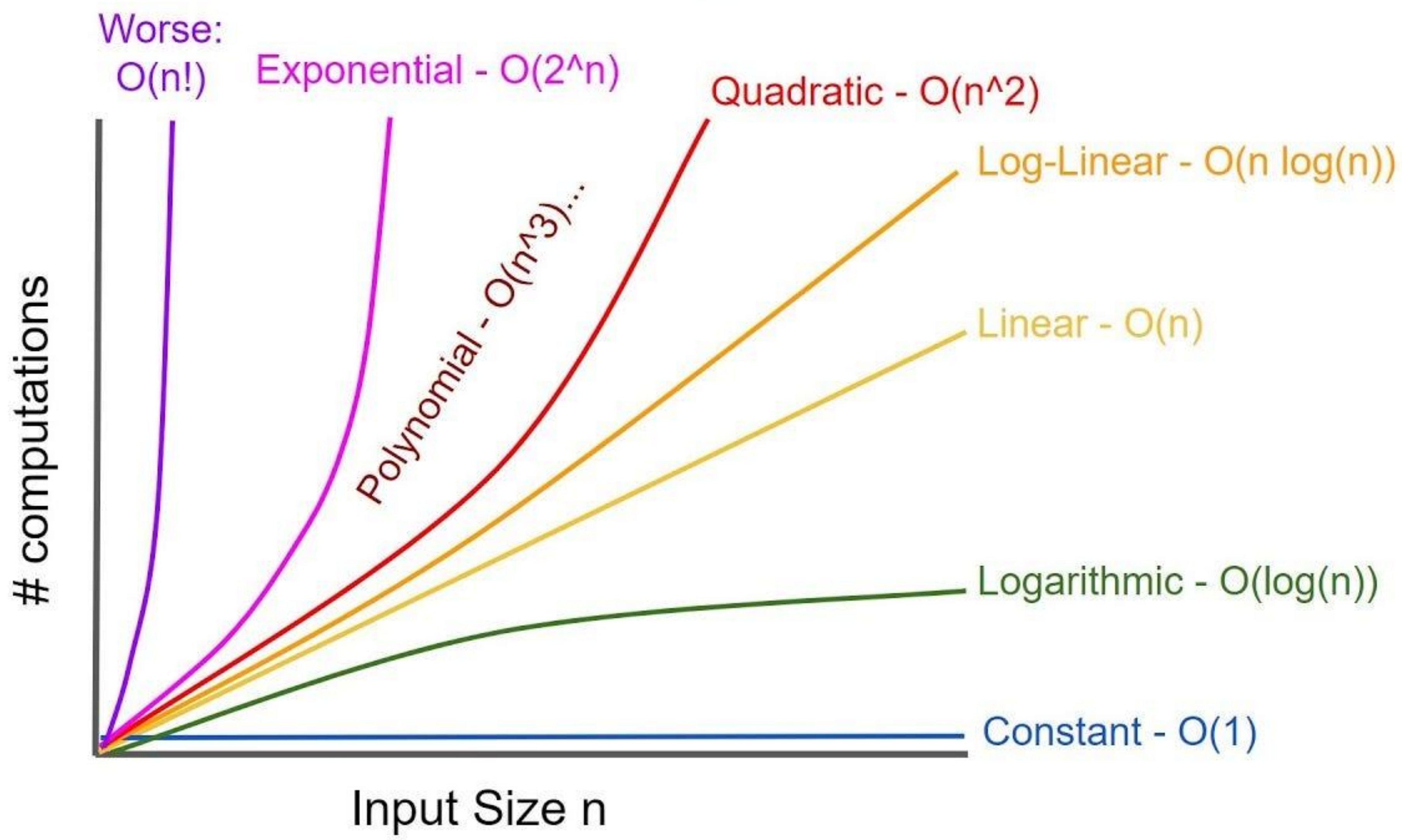
1	2	9	10	11	15	18	21	27	31	41	43	50

```java
private static int binary_search(int[] array, int n) {
		int low = 0;
		int high = array.length - 1;
		while(low <= high) {
			int mid = (low + high) / 2;
			if(n == array[mid]) {
				return mid;
			}
			else if(n > array[mid]) {
				low = mid + 1;
			}
			else {
				high = mid - 1;
			}
		}
		return -1;
}
```

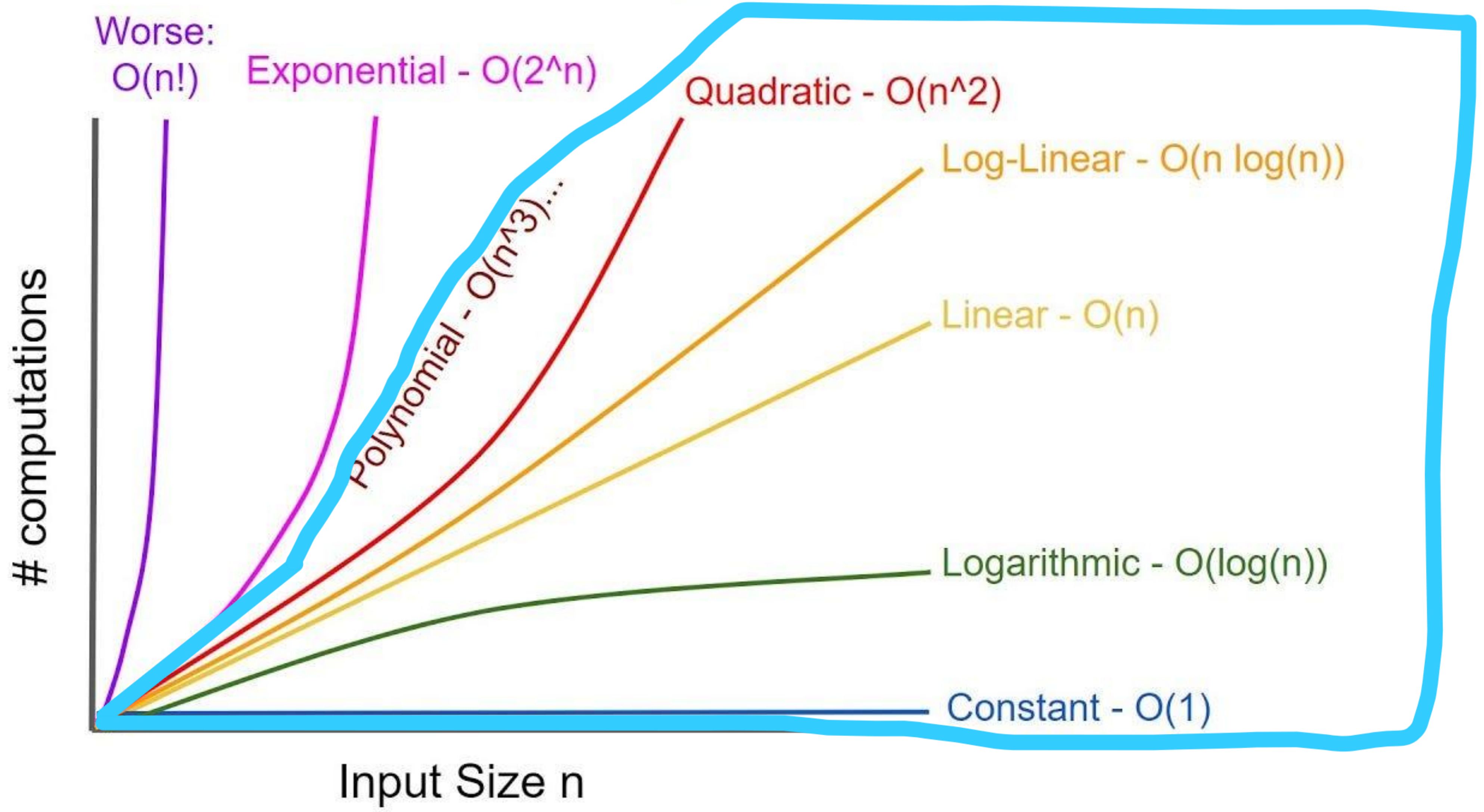
Array Size (N)	# of array spots checked
10	4
20	5
50	6
100	7
200	8
10000	14

Logarithmic Growth (logn)





Worse:
O(n!)
Exponential - O(2^n)
Quadratic - O(n^2)
Polynomial - O(n^3)...
Log-Linear - O(n log(n))
Linear - O(n)
Logarithmic - O(log(n))
Constant - O(1)
computations
Input Size n



Polynomial Time → "Acceptable"

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

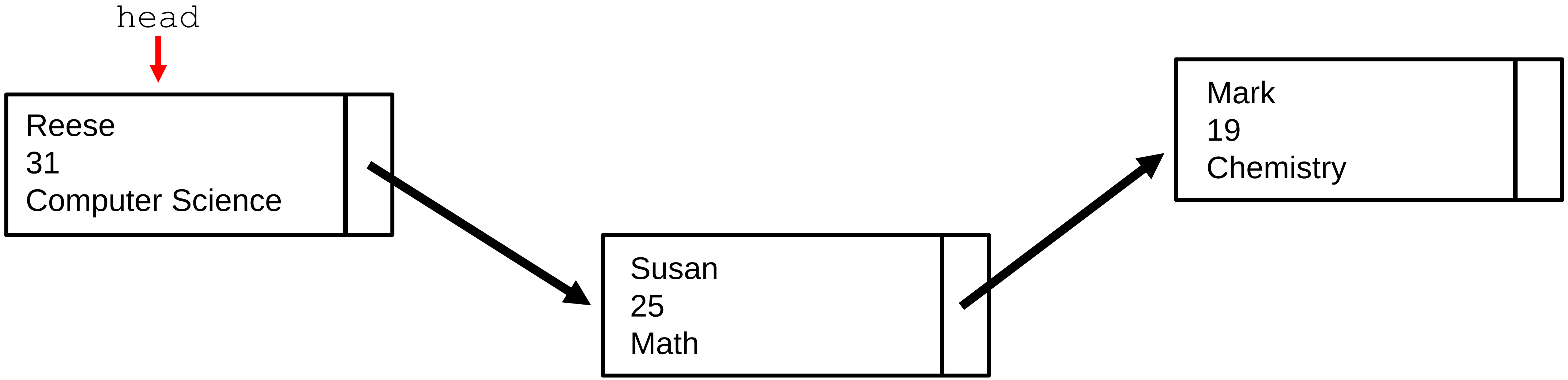
You can use the built-in Java array

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

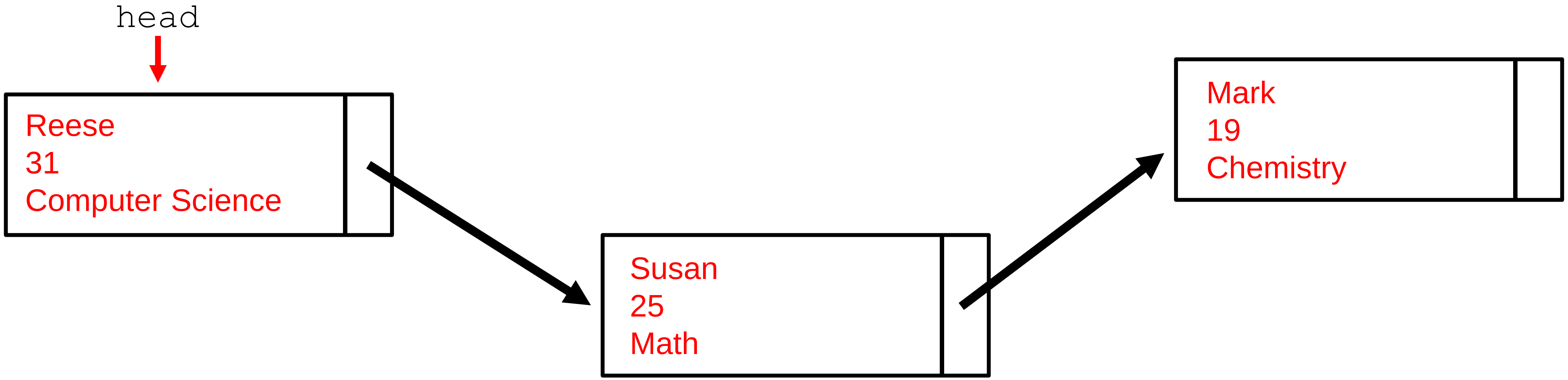
```
ArrayList<E>
```

```
LinkedList<E>
```

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



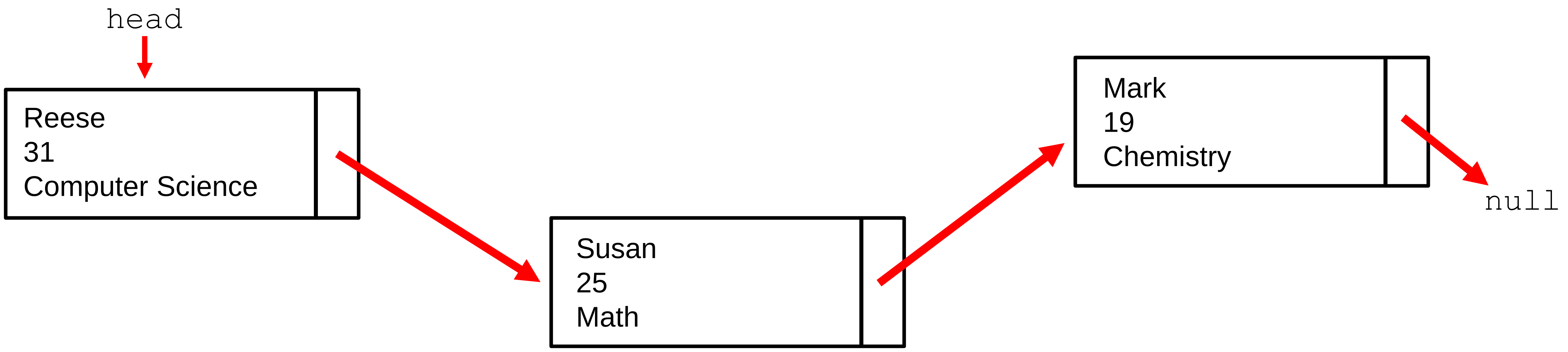
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Nodes consists of two parts:

1. Payload

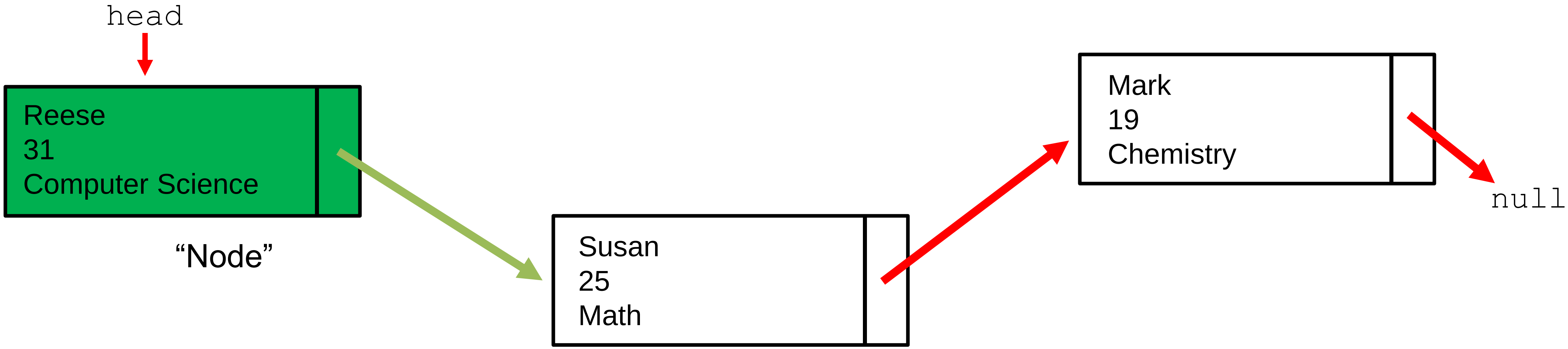
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



Nodes consists of two parts:

1. Payload
2. Pointer to next node

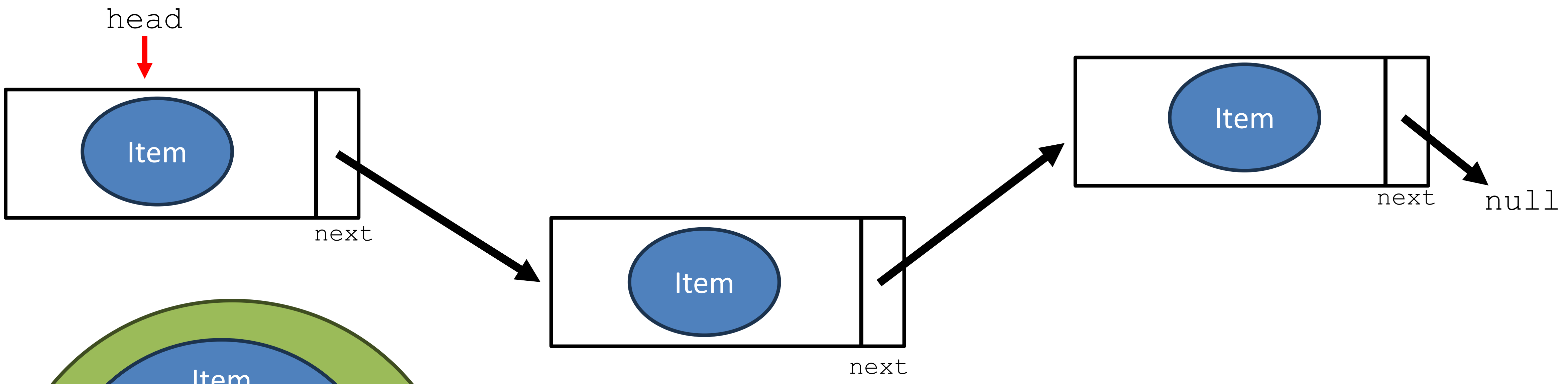
A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)

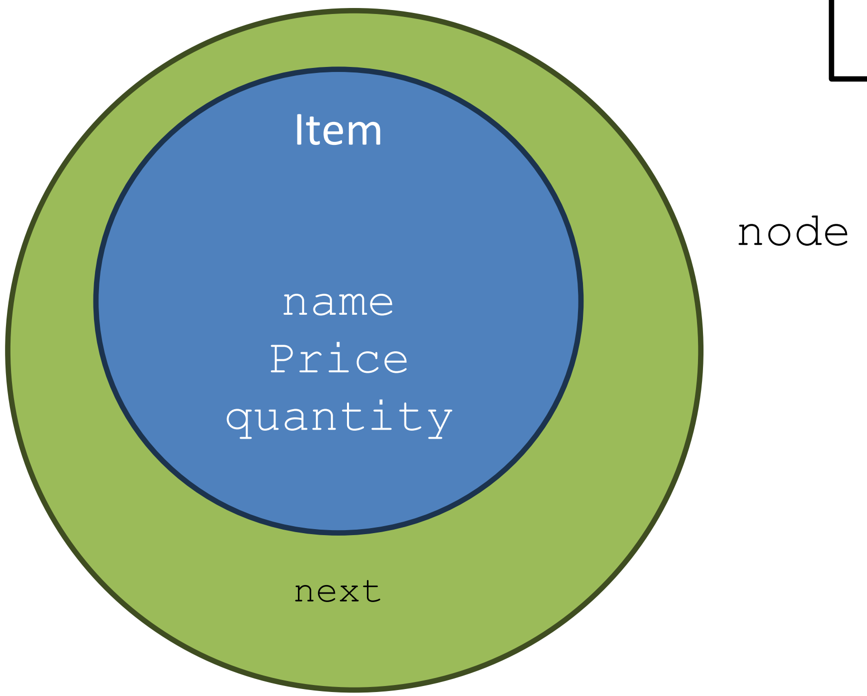


Nodes consists of two parts:

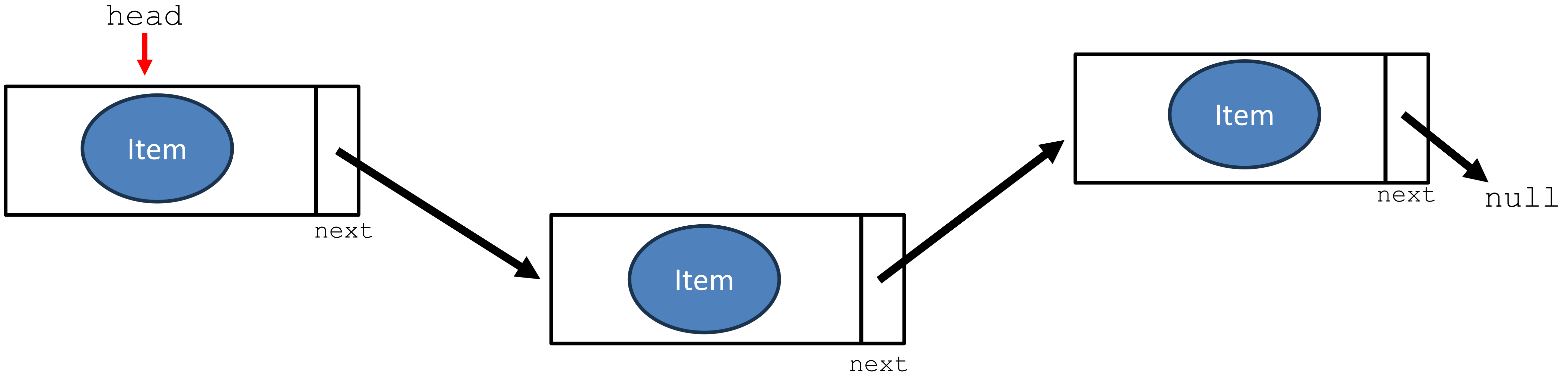
1. Payload
2. Pointer to next node

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



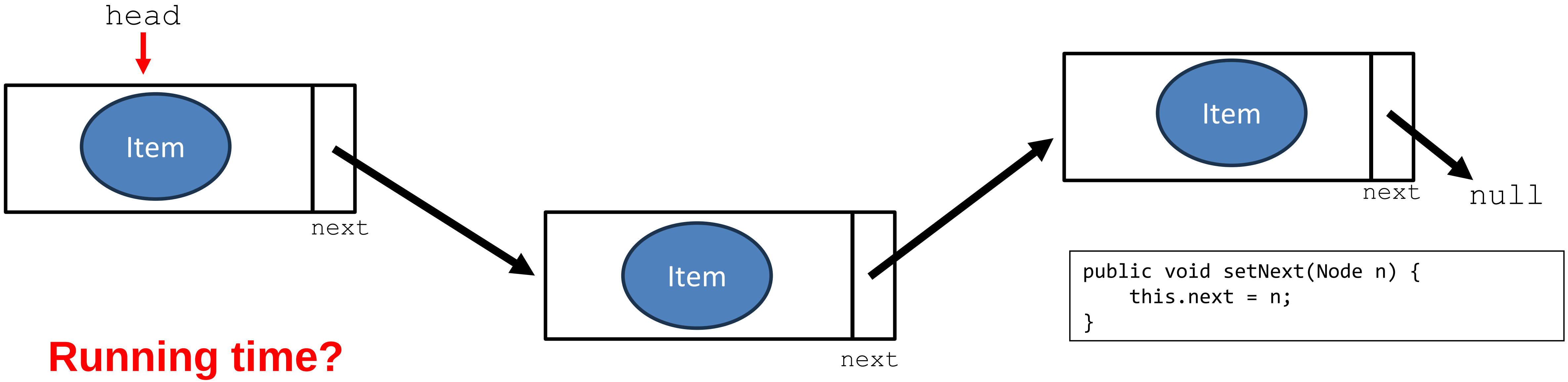


A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)




```java
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`)

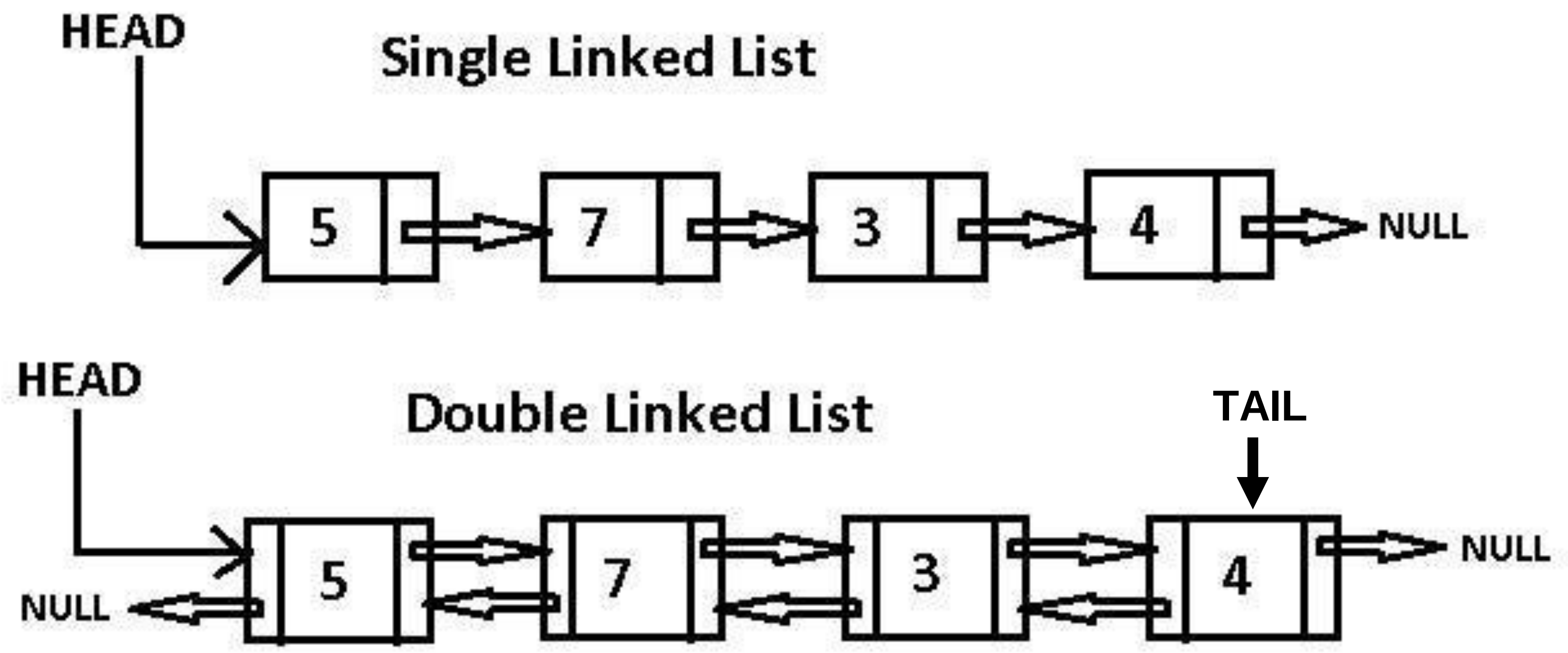



```
public void setNext(Node n) {
    this.next = n;
}
```

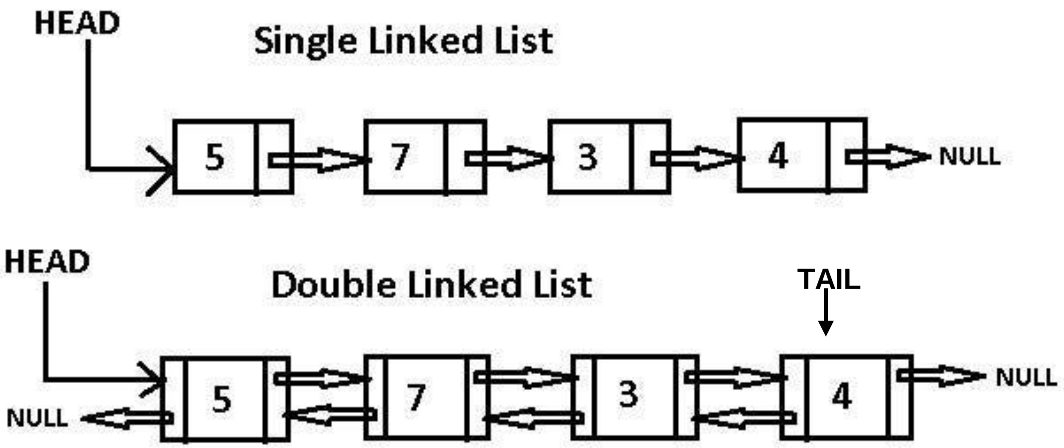
Running time?

```
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`)

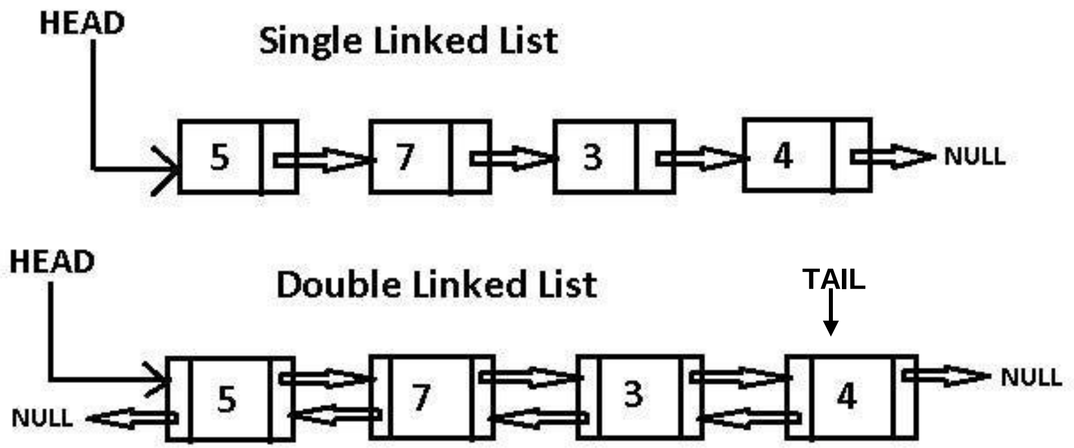


A **linked list** is a dynamic linear data structure that is a collection of data (`nodes`)



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;
```

Method Summary

Methods

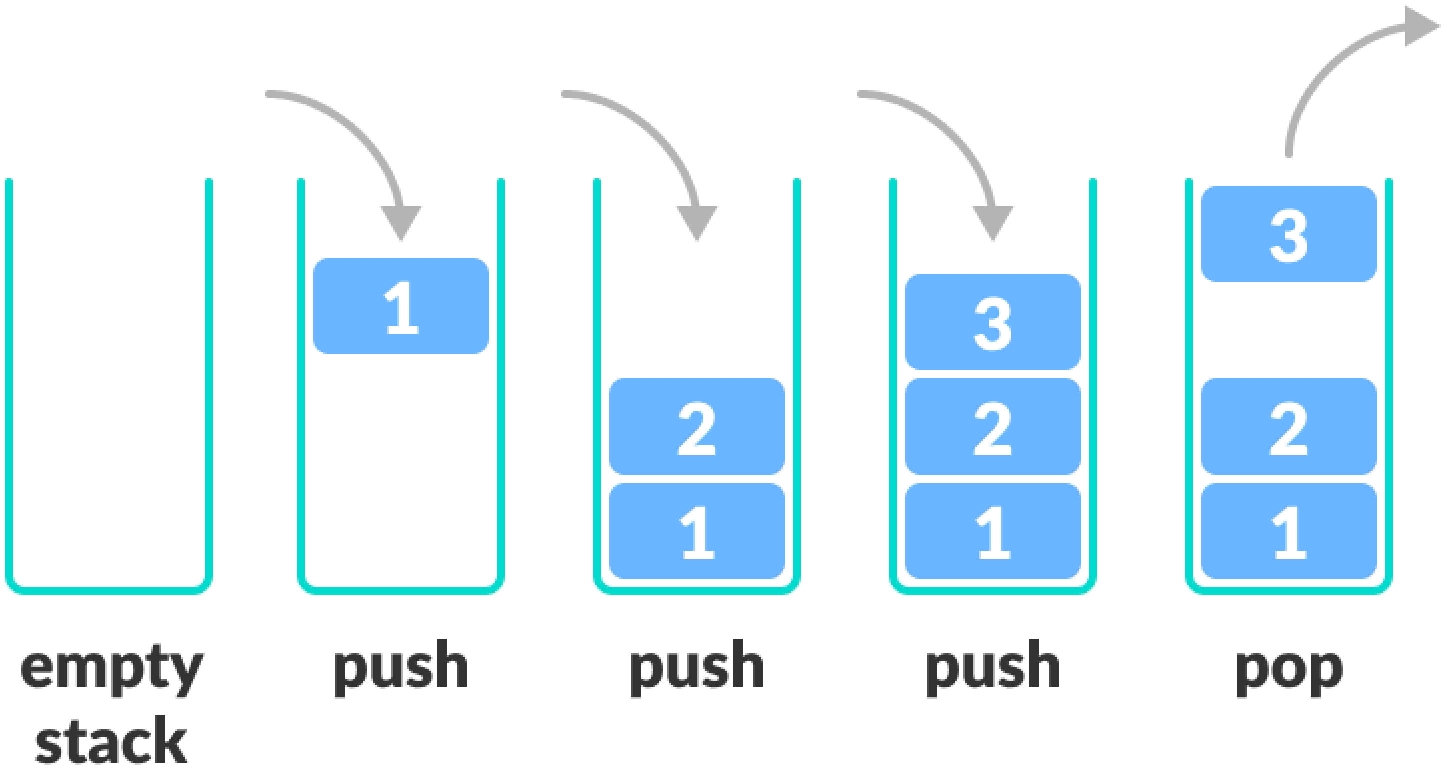
Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified 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.

```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);


    }

}
```

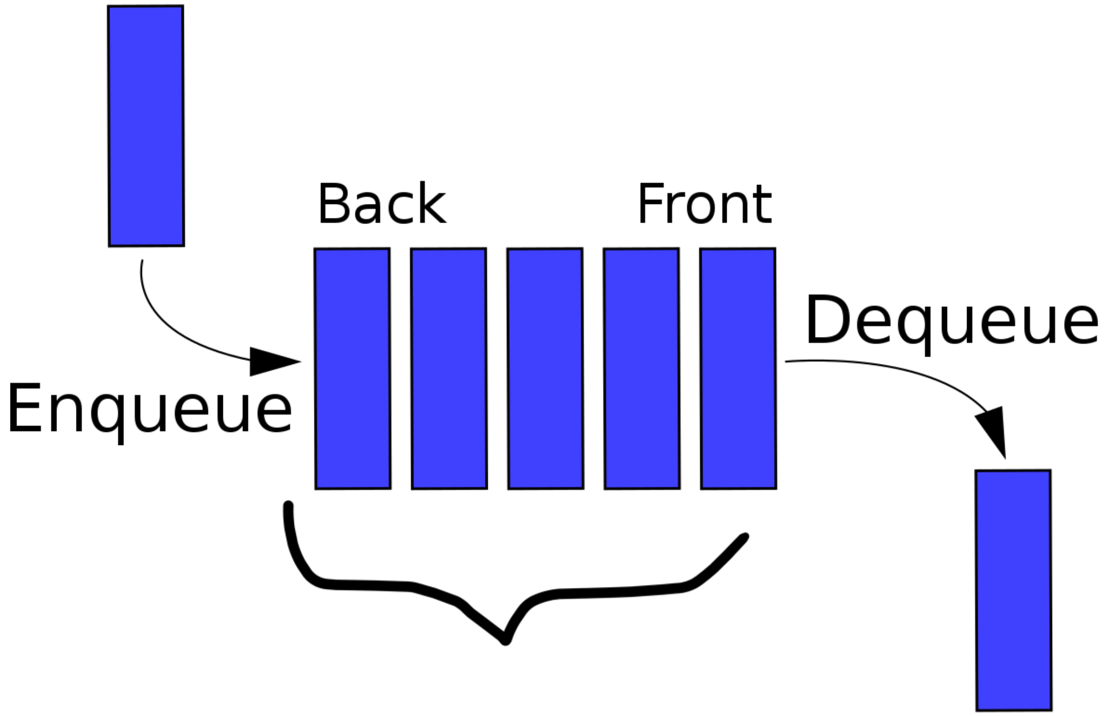
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

We can:

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

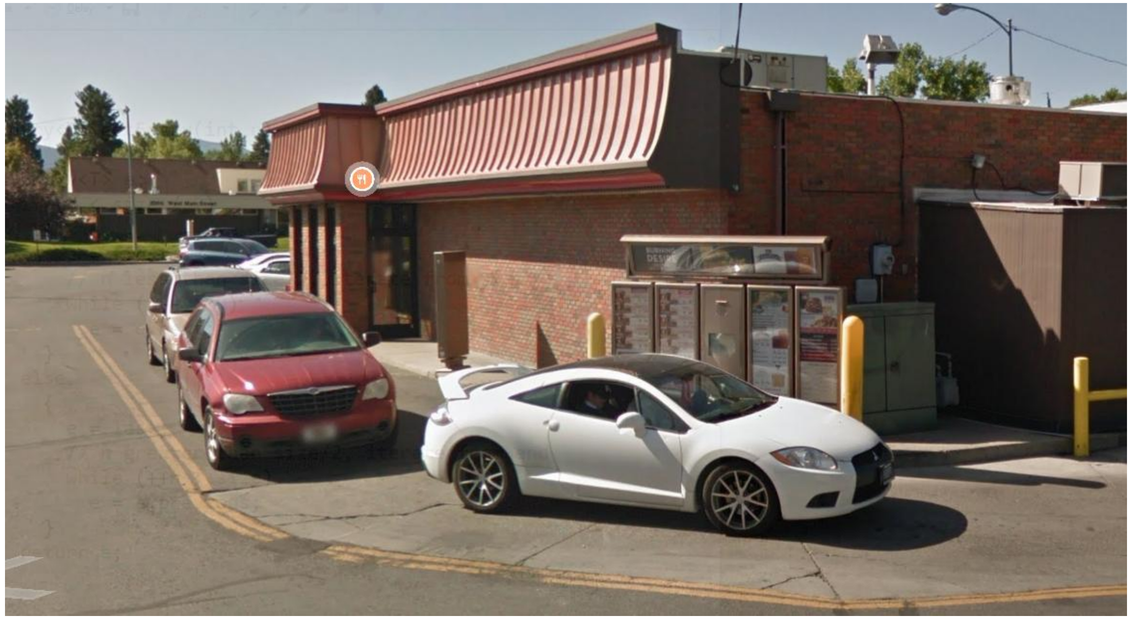


A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



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`

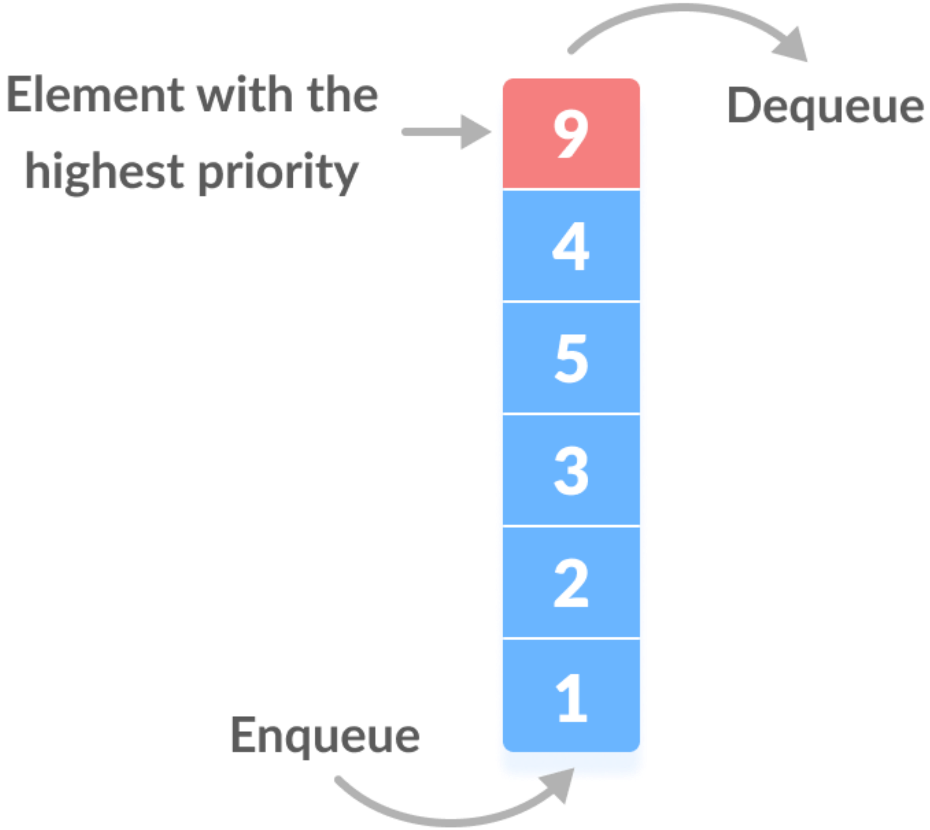
`java.util.Queue` is an interface. We cannot create a Queue object.
Instead, we create an instance of an object *that implements* this interface

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?)