

# Java Review

CSC220 // Java Programming 2

Today...



# Why Java?

## The 2017 Top Programming Languages

Python jumps to No. 1, and Swift enters the Top Ten

Posted 18 Jul 2017 | 19:00 GMT

By STEPHEN CASS

It's summertime here at *IEEE Spectrum*, and that means it's time for our fourth interactive ranking of the top programming languages. As with all attempts to rank the usage of different languages, we have to rely on various proxies for popularity. In our case, this means having data journalist [Nick Diakopoulos](#) mine and combine 12 metrics from 10 carefully chosen online sources to rank 48 languages. But where we really differ from other rankings is that our interactive allows you choose how those metrics are weighted when they are combined, letting you personalize the rankings to your needs.

We have a few preset weightings—a default setting that's designed with the typical *Spectrum* reader in mind, as well as settings that emphasize emerging languages, what employers are looking for, and what's hot in open source. You can also filter out industry sectors that don't interest you or create a completely customized ranking and make a comparison with a previous year.

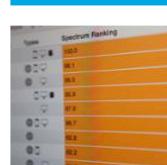
So what are the Top Ten Languages for the typical *Spectrum* reader?

Language Rank	Types	Spectrum Ranking
1. Python	🌐💻	100.0
2. C	💻🌐	99.7
3. Java	🌐💻🌐	99.5
4. C++	💻🌐🌐	97.1
5. C#	🌐💻🌐	87.7
6. R	💻	87.7
7. JavaScript	🌐💻	85.6

## The 2018 Top Programming Languages

Python extends its lead, and Assembly enters the Top Ten

By Stephen Cass



Explore the Interactive Rankings

Welcome to *IEEE Spectrum's* fifth annual interactive ranking of the top programming languages. Because no one can peer over the shoulders of every coder out there, anyone attempting to measure the popularity of computer languages must rely on proxy measures of relative popularity. In our case, this means combining metrics from multiple sources to rank 47 languages. But recognizing that different programmers have different needs and domains of interest, we've chosen not to blend all those metrics up into One Ranking to Rule Them All.

Instead, our interactive app lets you choose how these metrics are weighted when they are combined, so you can put an emphasis on what matters to you. (There's a [detailed description of our methods and sources](#) available.) We do include a default weighting, tuned to the interests of a typical IEEE member, and we offer some other presets that focus on things like what's *au courant* for open source projects. You can also apply filters that exclude languages primarily used in areas that aren't of interest to you, such as embedded or desktop environments. And you can see how things have changed by making comparisons with earlier years.

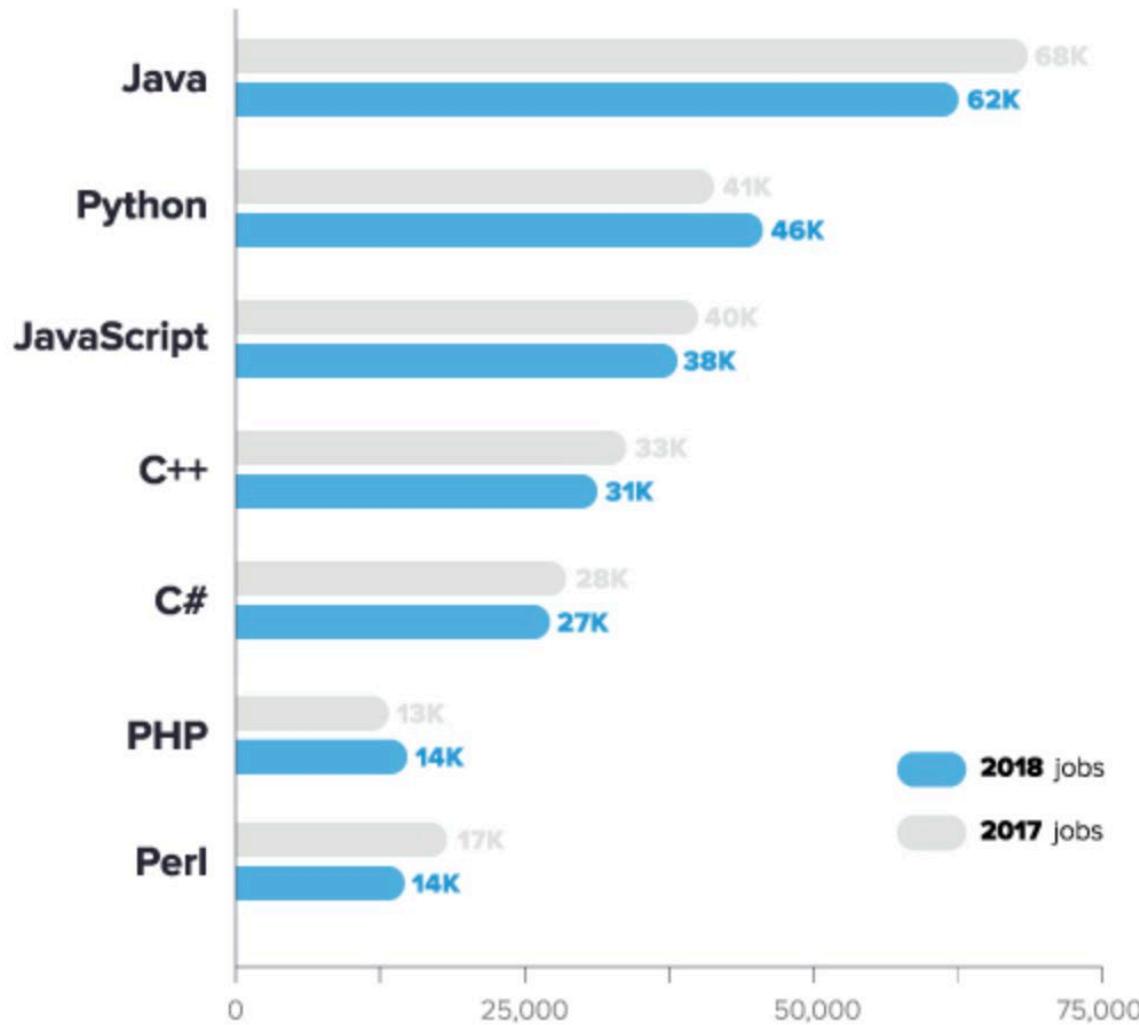
So what are the Top Ten Languages of 2018, as ranked for the typical IEEE member and *Spectrum* reader?

Language Rank	Types	Spectrum Ranking
1. Python	🌐💻	100.0
2. C++	💻🌐	98.4
3. C	💻🌐	98.2
4. Java	🌐💻🌐	97.5
5. C#	🌐💻🌐	89.8
6. PHP	🌐	85.4
7. R	💻	83.3
8. JavaScript	🌐💻	82.8
9. Go	🌐💻	76.7
10. Assembly	💻	74.5

Language Rank	Types	Spectrum Ranking
1. Python	   	100.0
2. C++	  	98.4
3. C	  	98.2
4. Java	  	97.5
5. C#	  	89.8
6. PHP		85.4
7. R		83.3
8. JavaScript	 	82.8
9. Go	 	76.7
10. Assembly		74.5

## Job postings containing top languages

Indeed.com - November, 17th 2017



Similar for 2019.

This week...

- **Types and variables (2.1 and 2.2)**
- **Control flow (2.3, 4.1)**
- **Logical operators (5.3)**
- **Parameter passing (3.1, 3.2)**
- **Reference types (3.3, 7.1, 7.3)**

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**
- What is a type?

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**
- What is a type?
  - a basic building block in a programming language
  - what kind of data a variable holds, and what operations can be performed on it

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**
- What is a type?
  - a basic building block in a programming language
  - what kind of data a variable holds, and what operations can be performed on it
- Java defines eight primitive types

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**
- What is a type?
  - a basic building block in a programming language
  - what kind of data a variable holds, and what operations can be performed on it
- Java defines eight primitive types
  - int, double, char, boolean, byte, short, long, float

# Variables

- A **variable** is a piece of data in memory with:
  - An identifier (name)
  - A **type**
- What is a type?
  - a basic building block in a programming language
  - what kind of data a variable holds, and what operations can be performed on it
- Java defines eight primitive types
  - int, double, char, boolean, byte, short, long, float
  - each primitive type can hold a single value
    - 12, 2.64, 'r', true

# Declaration & initialization

- **declaring a variable is stating that it exists**
  - assigns the variable a type and name

```
boolean areWeThereYet;
```

# Declaration & initialization

- **declaring a variable is stating that it exists**
  - assigns the variable a type and name  
`boolean areWeThereYet;`
- **initializing a variable gives it an initial value, and is often combined with declaring**  
`boolean areWeThereYet = false;`

# Declaration & initialization

- **declaring a variable is stating that it exists**
  - assigns the variable a type and name  
`boolean areWeThereYet;`
- **initializing a variable gives it an initial value, and is often combined with declaring**  
`boolean areWeThereYet = false;`
- **variables declared as final are constant and cannot be changed after initialization**  
`final int theMeaningOfLife = 42;`

# Assignment

- after a variable has been declared we can assign it a new value with =

```
areWeThereYet = true;
```

# Assignment

- after a variable has been declared we can assign it a new value with =

```
areWeThereYet = true;
```

- we can use arithmetic expressions with an assignment

```
age = currentYear - birthYear;
```

# Arithmetic operations

- **explicitly supported on primitive types**
  - binary operators
  - unary operators

# Arithmetic operations

- **explicitly supported on primitive types**
  - binary operators
    - + , - , \* , / , %
  - unary operators

# Arithmetic operations

- **explicitly supported on primitive types**

- binary operators

`+, -, *, /, %`

- unary operators

`++` (increment), `--` (decrement)

# Arithmetic operations

- **explicitly supported on primitive types**
  - binary operators
    - + , - , \* , / , %
  - unary operators
    - ++ (increment) , -- (decrement)
- **Java follows common order-of-operation rules**

# Arithmetic operations

- **explicitly supported on primitive types**
  - binary operators
    - + , - , \* , / , %
  - unary operators
    - ++ (increment) , -- (decrement)
- **Java follows common order-of-operation rules**

unary ops : highest

# Arithmetic operations

- **explicitly supported on primitive types**

- binary operators

`+, -, *, /, %`

- unary operators

`++ (increment), -- (decrement)`

- **Java follows common order-of-operation rules**

unary ops : highest

`*, /, %` : high

# Arithmetic operations

- **explicitly supported on primitive types**

- binary operators

`+, -, *, /, %`

- unary operators

`++ (increment), -- (decrement)`

- **Java follows common order-of-operation rules**

unary ops : highest

`*, /, %` : high

`+, -` : low

# Arithmetic operations

- **explicitly supported on primitive types**

- binary operators

`+, -, *, /, %`

- unary operators

`++ (increment), -- (decrement)`

- **Java follows common order-of-operation rules**

unary ops : highest

`*, /, %` : high

`+, -` : low

`=` : lowest

# Arithmetic operations

- explicitly supported on primitive types

- binary operators

+ , - , \* , / , %

- unary operators

++ (increment) , -- (decrement)

- Java follows common order-of-operation rules

unary ops : highest

\* , / , % : high

+ , - : low

= : lowest

int a = 3;

int b = 4;

int c = 2;

System.out.println(a + b \*c);

?

# Arithmetic operations

- explicitly supported on primitive types

- binary operators

+ , - , \* , / , %

- unary operators

++ (increment) , -- (decrement)

- Java follows common order-of-operation rules

unary ops : highest

\* , / , % : high

+ , - : low

= : lowest

int a = 3;

int b = 4;

int c = 2;

System.out.println(a + b \*c);

11

# Operator Precedence

Operator	Description	Level
[ ]	access array element	
.	access object member	
()	invoke a method	1
++	post-increment	
--	post-decrement	
++	pre-increment	
--	pre-decrement	
+	unary plus	
-	unary minus	
!	logical NOT	
~	bitwise NOT	
*		
/	multiplicative	3
%		
+ -	additive	
+	string concatenation	4
< <=		
> >=	relational type comparison	5

Source: <http://introcs.cs.princeton.edu/java/11precedence/>

Another nice example mentioned by you...

```
int i = 3;  
int j = i++ + 4;  
System.out.println(i + " " + j);
```

---

?

Another nice example mentioned by you...

```
int i = 3;  
int j = i++ + 4;  
System.out.println(i + " " + j);
```

---

4              7

# What about?

```
int x = 10;
```

```
int y = 20;
```

```
int z = ++x * y--;
```

# What about?

```
int x = 10;
```

```
int y = 20;
```

```
int z = ++x * y--;
```

```
System.out.println(x) ← ?
```

```
System.out.println(y) ← ?
```

```
System.out.println(z) ← ?
```

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

# Type conversion

- **widening conversions** convert data to another type that has the same or more bits of storage

# Type conversion

- **widening conversions** convert data to another type that has the same or more bits of storage
- **narrowing conversions** convert data to another type that has fewer bits of storage, possibly losing information

# Type conversion

- **widening conversions** convert data to another type that has the same or more bits of storage

short → int  
int → long

- **narrowing conversions** convert data to another type that has fewer bits of storage, possibly losing information

# Type conversion

- **widening conversions** convert data to another type that has the same or more bits of storage

short → int

int → long

- **narrowing conversions** convert data to another type that has fewer bits of storage, possibly losing information

double → float

float → int

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

```
2.2 * 2
```

```
1.0 / 2
```

```
double x = 2;
```

```
"count = " + 4
```

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

2.2 \* 2      **evaluates to 4.4**

1.0 / 2

double x = 2;

“count = “ + 4

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

2.2 \* 2      **evaluates to 4.4**

1.0 / 2      **evaluates to 0.5**

double x = 2;

“count = “ + 4

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

2.2 \* 2      **evaluates to 4.4**

1.0 / 2      **evaluates to 0.5**

double x = 2;      **assigns 2.0 to x**

“count = “ + 4

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

2.2 \* 2      **evaluates to 4.4**

1.0 / 2      **evaluates to 0.5**

double x = 2;      **assigns 2.0 to x**

“count = ” + 4      **evaluates to “count = 4”**

# Type conversion

- java uses widening conversion when an operator is applied to operands of different types (called **promotion**)

2.2 \* 2

**evaluates to 4.4**

1.0 / 2

**evaluates to 0.5**

double x = 2;

**assigns 2.0 to x**

“count = ” + 4

**evaluates to “count = 4”**

↑  
concatenate

Integer converts it to an Integer: new Int(4)

Converted to a String: Integer.toString()

Two Strings were concatenated

# Mixing types

- conversions are done on one operator at a time in the order the operators are evaluated

# Mixing types

- conversions are done on one operator at a time in the order the operators are evaluated

```
3 / 2 * 3.0 + 8 / 3
```

# Mixing types

- conversions are done on one operator at a time in the order the operators are evaluated

```
3 / 2 * 3.0 + 8 / 3      5.0
```

# Mixing types

- conversions are done on one operator at a time in the order the operators are evaluated

3 / 2 \* 3.0 + 8 / 5      **5.0**

2.0 \* 4 / 5 + 6 / 4.0

# Mixing types

- conversions are done on one operator at a time in the order the operators are evaluated

3 / 2 \* 3.0 + 8 / 5      **5.0**

2.0 \* 4 / 5 + 6 / 4.0      **3.1**

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

```
1 + "x" + 4
```

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

```
1 + "x" + 4      "1x4"
```

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      "1x4"

"2+3=" + 2 + 3

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

"2\*3=" + 2 \* 3

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

"2\*3=" + 2 \* 3      **"2\*3=6"**

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

"2\*3=" + 2 \* 3      **"2\*3=6"**

4 - 1 + "x"

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

"2\*3=" + 2 \* 3      **"2\*3=6"**

4 - 1 + "x"      **"3x"**

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      "1**x**4"

"2+3=" + 2 + 3      "**2+3=23**"

1 + 2 + "3"      "**33**"

"2\*3=" + 2 \* 3      "**2\*3=6**"

4 - 1 + "x"      "**3x**"

"x" + 4 - 1

# Mixing types

- String concatenation has the same precedence as +- and is evaluated left to right

1 + "x" + 4      **"1x4"**

"2+3=" + 2 + 3      **"2+3=23"**

1 + 2 + "3"      **"33"**

"2\*3=" + 2 \* 3      **"2\*3=6"**

4 - 1 + "x"      **"3x"**

"x" + 4 - 1      **error**

# Type casting

- **Type casting** tells Java to convert one type to another
  - Convert **int** to **double** to force floating-point division

```
Double average = (double) 12 / 5;
```

- Truncate a **double** to an **int**

```
int feet = (int) (25.0 / 12.0);
```

# Type casting

- **Type casting** tells Java to convert one type to another
  - Convert `int` to `double` to force floating-point division

```
Double average = (double) 12 / 5;
```

How about this?!

## Use `toString` Method

```
double d = 12.5;  
String s1 = Double.toString(d);
```

```
int i = 12;  
String s2 = Integer.toString(i);
```

```
System.out.println(s1); ----- 12.5  
System.out.println(s2); ----- 12
```

# Casting to String

```
12  public static void main(String[] args) {  
13  
14      String s = (String) 12;           Cannot cast from int to String  
15  
16      System.out.println(s);  
17  
18  }
```

What to do?!

# The Java™ Tutorials

•<https://docs.oracle.com/javase/tutorial/java/>

The screenshot shows the Oracle Java Documentation website. At the top left is the Oracle logo and the Java Documentation link. The top right has links for "Download Ebooks", "Download JDK", and "Search Java Tutorials". Below the header, the title "The Java™ Tutorials" is displayed, along with a "Home Page" link on the right. Underneath, there's a breadcrumb trail with links for "« Previous", "Trail", and "Next »". The main content area is titled "Trail: Learning the Java Language". It lists several lessons with icons and brief descriptions:

- Object-Oriented Programming Concepts**: Teaches core concepts like objects, messages, classes, and inheritance.
- Language Basics**: Describes traditional features like variables, arrays, data types, operators, and control flow.
- Classes and Objects**: Describes how to write classes and create objects.
- Annotations**: Describes annotations as metadata for the compiler.
- Interfaces and Inheritance**: Describes interfaces, subclasses, and how they inherit from superclasses.
- Numbers and Strings**: Describes Number and String objects and data formatting.
- Generics**: Describes a feature for type safety and code detection.
- Packages**: Describes packages for organizing classes.

« Previous • Trail • Next »

[Home Page](#)

## Trail: Learning the Java Language

This trail covers the fundamentals of programming in the Java programming language.

**Object-Oriented Programming Concepts** teaches you the core concepts behind object-oriented programming: objects, messages, classes, and inheritance. This lesson ends by showing you how these concepts translate into code. Feel free to skip this lesson if you are already familiar with object-oriented programming.

**Language Basics** describes the traditional features of the language, including variables, arrays, data types, operators, and control flow.

**Classes and Objects** describes how to write the classes from which objects are created, and how to create and use the objects.

**Annotations** are a form of metadata and provide information for the compiler. This lesson describes where and how to use annotations in a program effectively.

**Interfaces and Inheritance** describes interfaces—what they are, why you would want to write one, and how to write one. This section also describes the way in which you can derive one class from another. That is, how a *subclass* can inherit fields and methods from a *superclass*. You will learn that all classes are derived from the `Object` class, and how to modify the methods that a subclass inherits from superclasses.

**Numbers and Strings** This lesson describes how to use `Number` and `String` objects. The lesson also shows you how to format data for output.

**Generics** are a powerful feature of the Java programming language. They improve the type safety of your code, making more of your bugs detectable at compile time.

**Packages** are a feature of the Java programming language that help you to organize and structure your classes and their relationships to one another.

« Previous • TOC • Next »

Your use of this page and all the material on pages under "The Java Tutorials" banner is subject to these [legal notices](#).

Problems with the examples? Try [Compiling and Running the Examples: FAQs](#).

Copyright © 1995, 2015 Oracle and/or its affiliates. All rights reserved.

Complaints? Compliments? Suggestions? [Give us your feedback](#).

# Assignment operators

- Basic assignment operation  
=
- Combined assignment/arithmetic operators  
+=, -=, \*=, /=
- Increment/decrement operators can be stand-alone statements

i++;

i--;

++i;

--i;

# Assignment operators

- Basic assignment operation
  - =
- Combined assignment/arithmetic operators
  - +=, -=, \*=, /=
- Increment/decrement operators can be stand-alone statements
  - i++;

```
int i = 3;  
i--;      int j = i++;  
++i;      System.out.println(i+" "+j);  
--i;
```

---

?

# Assignment operators

- Basic assignment operation
  - =
- Combined assignment/arithmetic operators
  - +=, -=, \*=, /=
- Increment/decrement operators can be stand-alone statements
  - i++;

```
int i = 3;  
i--;      int j = i++;  
++i;      System.out.println(i+" "+j);  
--i;
```

---

4 3

# Assignment operators

- Basic assignment operation
  - =
- Combined assignment/arithmetic operators
  - +=, -=, \*=, /=
- Increment/decrement operators can be stand-alone statements
  - i++;

```
int i = 3;  
i--;      int j = ++i;  
++i;      System.out.println(i+" "+j);  
--i;
```

---

?

# Assignment operators

- Basic assignment operation
  - =
- Combined assignment/arithmetic operators
  - +=, -=, \*=, /=
- Increment/decrement operators can be stand-alone statements
  - i++;

```
int i = 3;  
i--;      int j = ++i;  
++i;      System.out.println(i+" "+j);  
--i;
```

---

4 4

# Control Flow

- **Control flow** determines how programs make decisions about what to do, and how many times to do it

Looping: `for`, `while`, `do-while`

Decision making: `if-else`, `switch-case`

Jumping: `break`, `continue`, `return`

Exceptions: `try-catch`, `throw`

# Loops

- Perfect to automate redundant task

# Loops

- Perfect to automate redundant task
- Let's say we want to generate 5 random numbers.

```
System.out.println(Math.random());  
System.out.println(Math.random());  
System.out.println(Math.random());  
System.out.println(Math.random());  
System.out.println(Math.random());
```

# Loops

- Perfect to automate redundant task
- Let's say we want to generate 5 random numbers.
- Or...

```
for (int i = 1; i <= 5; i++){  
    System.out.println(Math.random());  
}
```

# Switch statements

- Similar to an if-else-if statement

```
switch (integer expression)
{
    case <integer literal>:
        list of statements...
    case <integer literal>:
        ...
}
```

# Switch statements

- Execution begins on the \_\_\_\_\_ case that matches the value of the switch variable
- Execution continues until \_\_\_\_\_ is reached

# Switch statements

- Execution begins on the **first** case that matches the value of the switch variable
- Execution continues until **break** is reached

# Switch statements

- Execution begins on the **first** case that matches the value of the switch variable
- Execution continues until **break** is reached
  - Even continues through other cases!
  - Usually want a break after every case

# Switch statements

- Execution begins on the **first** case that matches the value of the switch variable
- Execution continues until **break** is reached
  - Even continues through other cases!
  - Usually want a break after every case
- Switches can use the **default** keyword
  - If no cases were hit, execute the **default** case

# Switch example

```
int place = 1;

switch (place){
    case 1:
        System.out.println("gold medal");
        break;
    case 2:
        System.out.println("silver medal");
        break;
    case 3:
        System.out.println("bronze medal");
        break;
    case 4:
        System.out.println("no medal! go enjoy Rio");
        break;
    default:
        System.out.println("not even participated...");
        break;
}
```

Next Time...

- Java Review
  - Logical operation
  - Parameter passing
  - Reference types

# Java Review

CSC220 // Java Programming 2 //

# Today...

- Logical operators (5.3)
- Parameter passing (3.1, 3.2)
- Reference types (3.3, 7.1, 7.3)  
reference  
types (3.3, 7.1, 7.3)

# Logical Operators

# Relational and logical ops

- > , < , >= , <= , == , !=

# Relational and logical ops

- `>` ,    `<` ,    `>=` ,    `<=` ,    `==` ,    `!=`
- Results are always boolean

# Relational and logicalops

- `>` ,    `<` ,    `>=` ,    `<=` ,    `==` ,    `!=`
- Results are always boolean
- Relational ops supported for:

# Relational and logical ops

- `>` ,    `<` ,    `>=` ,    `<=` ,    `==` ,    `!=`
- Results are always boolean
- Relational ops supported for:
  - numbers
  - character types

# Relational and logical ops

- > , < , >= , <= , == , !=

- Results are always boolean

- Relational ops supported for:

- numbers

- character types

- Logical ops supported for boolean

- && , || , ! , == , !=

# Relational and logical ops precedence

- All lower than arithmetic operators

Operator	Description	Level
[ ] . ( ) ++ --	access array element access object member invoke a method post-increment post-decrement	1
++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2
*		
/		
%	multiplicative	3
+ - +	additive string concatenation	4
< <= > >=	relational type comparison	5

# Relational and logical ops precedence

- All lower than arithmetic operators

`>, <, >=, <= : highest`

# Relational and logical ops precedence

- All lower than arithmetic operators

`>, <, >=, <=` : highest

`==, !=` : high

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <= : highest

==, != : high

&& : low

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <=	:	highest
==, !=	:	high
&&	:	low
	:	lowest

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <= : highest

==, != : high

&& : low

|| : lowest

n >= 10 && n <= 99

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <=	:	highest
==, !=	:	high
&&	:	low
	:	lowest

```
int i = 3;  
int j = 4;  
  
if (++i == j)  
    System.out.println("You got it :)");
```

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <= : highest

==, != : high

&& : low

|| : lowest

```
int i = 3;
```

```
int j = 4;
```

```
if (i++ == j)
```

```
    System.out.println("i = 4");
```

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <=	:	highest
==, !=	:	high
&&	:	low
	:	lowest

```
int i = 3;  
int j = 4;  
  
if (++i == j)  
    System.out.println("i = 4");
```

# Relational and logical ops precedence

- All lower than arithmetic operators

>, <, >=, <=	:	highest
==, !=	:	high
&&	:	low
	:	lowest

```
int i = 3;  
int j = 4;
```

```
if (i+1 == j)  
    System.out.println("i = " + i);
```

# Parameter Passing (aka Methods and stuff)

# Parameter passing

- What does the following piece of code return?

```
static public void modifyInt(int i ){  
    ++i;  
}  
  
public static void main(String[] args) {  
  
    int i = 4;  
    modifyInt(i);  
    System.out.println(i);  
}
```

# Parameter passing

- Formal parameter vs. Actual parameter

```
static public void modifyInt(int i ){  
    ++i;  
}  
  
public static void main(String[] args) {  
  
    int i = 4;  
    modifyInt(i);  
    System.out.println(i);  
}
```

# Parameter passing

- What does the following piece of code return?

```
static public void modifyInt(int i ){  
    ++i;  
}
```

```
public static void main(String[] args) {  
  
    int i = 4;  
    modifyInt(i);  
    System.out.println(i); —————→ 4  
}
```

# Parameter passing

- Java uses **call-by-value** parameter passing
  - i.e., a copy is created

```
static public void modifyInt(int i ){  
    ++i;  
}  
  
public static void main(String[] args) {  
  
    int i = 4;  
    modifyInt(i);  
    System.out.println(i);  
}
```

# Parameter passing

- Java uses **call-by-value** parameter passing
  - i.e., a copy is created

```
static public void modifyInt(int i ){  
    ++i;  
}
```

```
public static void main(String[] args) {  
  
    int i = 4;  
    modifyInt(i);  
    System.out.println(i);  
}
```

# Parameter passing

- Java uses **call-by-value** parameter passing
  - i.e., a copy is created

```
static public int modifyInt(int i ){  
    return ++i;  
}
```

```
public static void main(String[] args) {  
  
    int i = 4;  
    int j = modifyInt(i);  
    System.out.println(i + " " + j);
```

# Parameter passing

- Java uses **call-by-value** parameter passing
  - i.e., a copy is created

```
static public int modifyInt(int i ){  
    return ++i;  
}
```

```
public static void main(String[] args) {  
  
    int i = 4;  
    int j = modifyInt(i);  
    System.out.println(i + " " + j);  
}
```

# main

- When a program runs, the `main` method is invoked

```
public static void main (String[] args)
```

- The parameters of `main` can be set using command-line arguments
- Think of the `main` method as where the “main stuff” happens. i.e. where the instructions/directives go

# Reference Types

# Reference type

- Recall Java primitive types
  - byte, short, int, long, float, double, char, boolean

# Reference type

- Recall Java primitive types
  - byte, short, int, long, float, double, char, boolean
- All non-primitive types are **reference types**
  - Arrays
  - String
  - ...

# Reference type

- It is all about how variables are stored in the memory

# Reference type

- It is all about how variables are stored in the memory

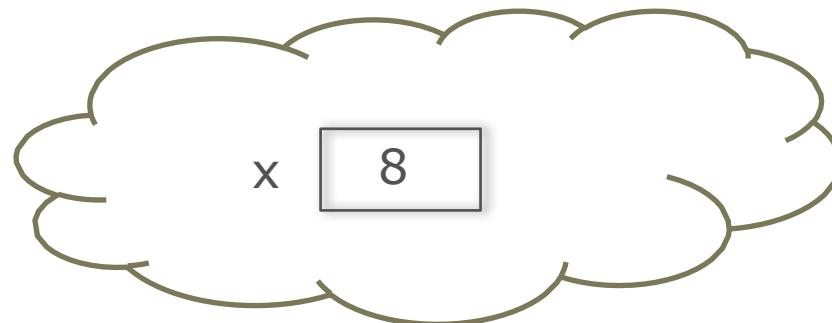
```
int x = 8;
```

# Reference type

- It is all about how variables are stored in the memory

```
int x = 8;
```

- The variable stores the actual data.

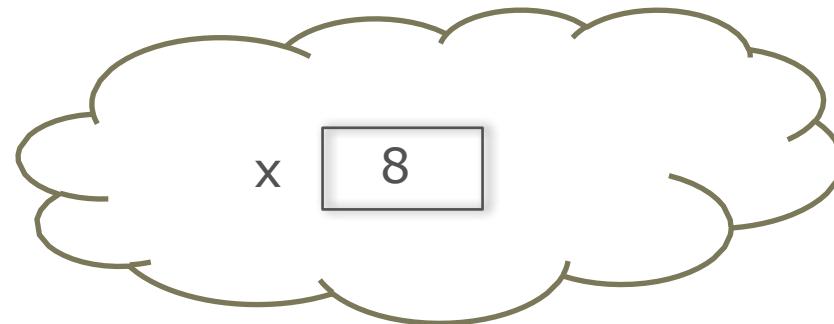


# Reference type

- It is all about how variables are stored in the memory

```
int x = 8;
```

- The variable stores the actual data.



- It will look different for reference types.

# Reference type

- Reference types, since they are “Objects”, are stored by “Reference”
- Basically stored by the address in memory of where it’s stored, not the actual object itself

# Arrays

- An array is a mechanism for storing a collection of identically types entities
- The [ ] operator indexes an array, accessing an individual entity – bounds checking is performed automatically

# Arrays

- An array is a mechanism for storing a collection of identically types entities
- The [ ] operator indexes an array, accessing an individual entity – bounds checking is performed automatically
- By default, array elements are initialized 0 (primitive types)

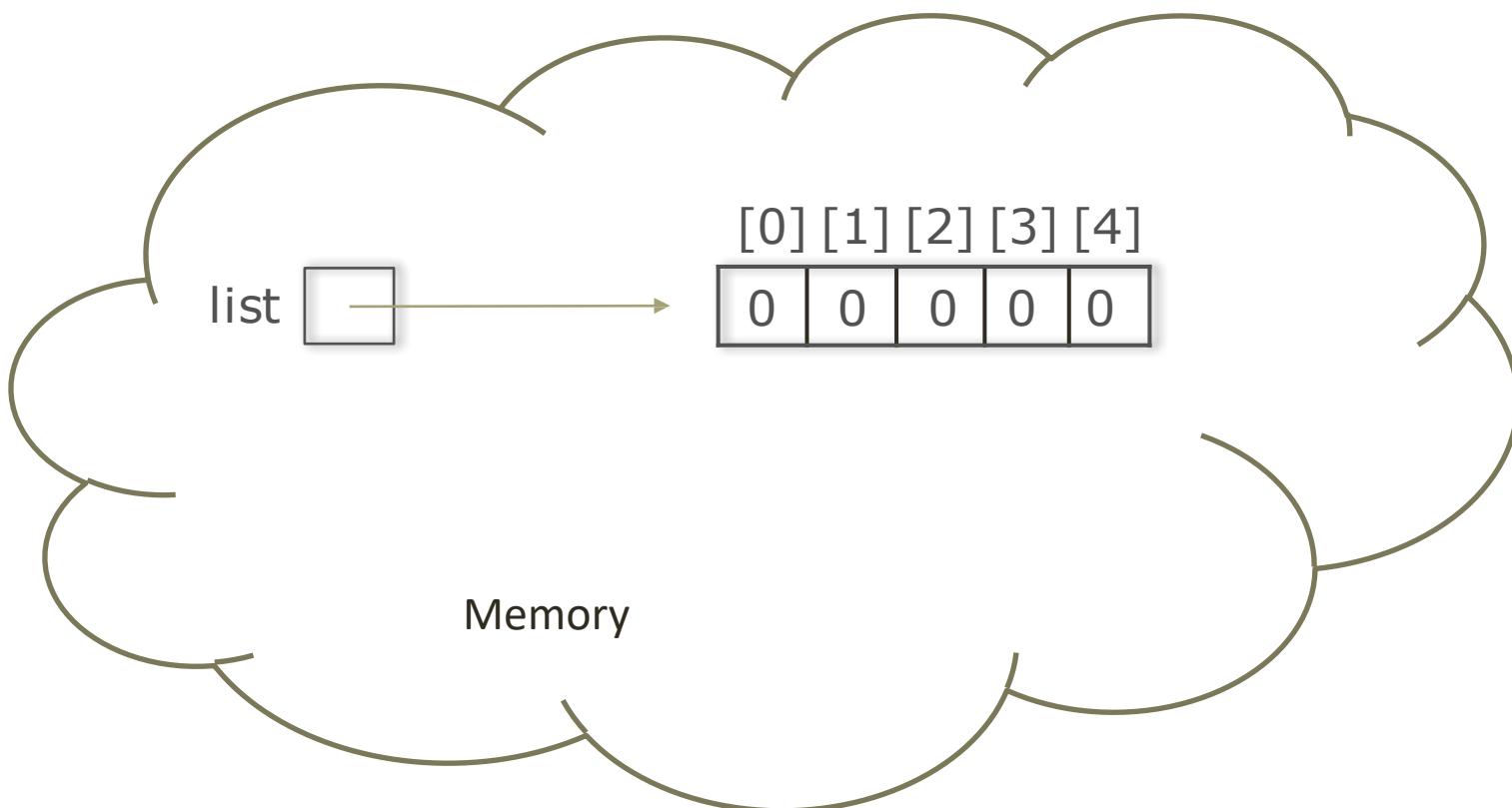
```
double primArray = new double[3];  
double[] primArray = {2.14, 2.2, -9.8};
```

# Arrays

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

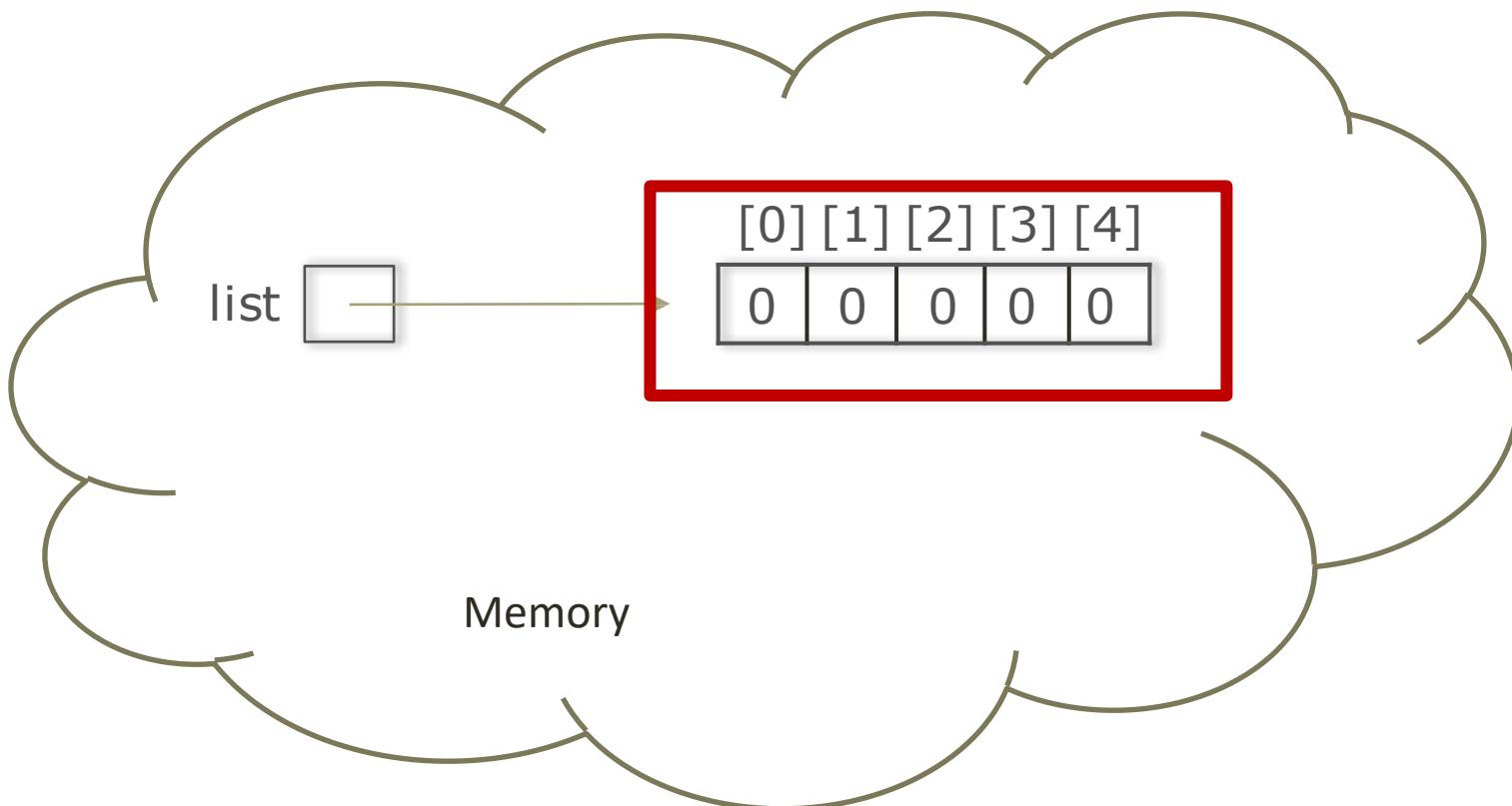
# Arrays

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



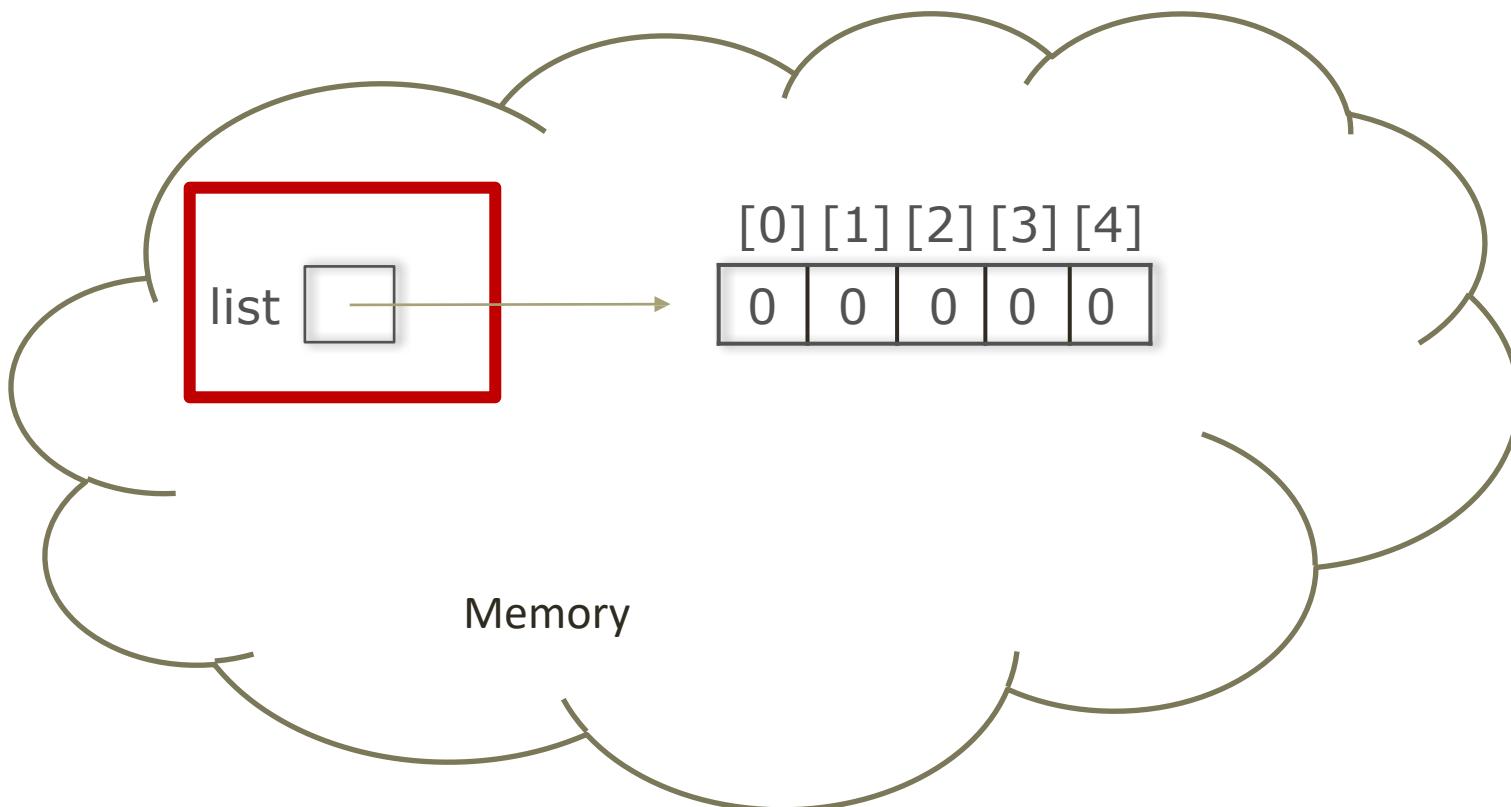
# Arrays

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



# Arrays

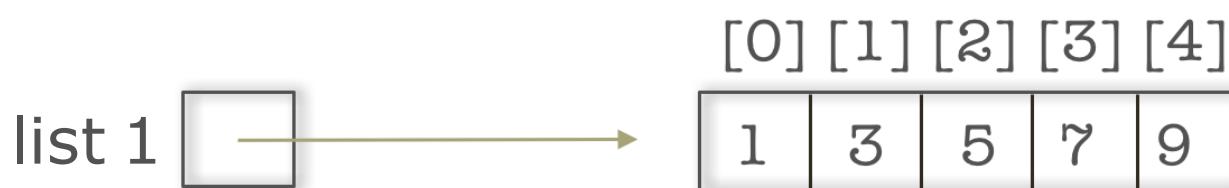
```
int[] list = new int[5];
```



# Reference Semantics

```
int[] list1 = new int[5];
```

```
for (int i = 0; i < list1.length; i++)
    list1[i] = 2 * i + 1;
```



# Reference Semantics

```
int[] list1 = new int[5];
```

```
for (int i = 0; i < list1.length; i++)
    list1[i] = 2 * i + 1;
```

```
int[] list2 = list1;
```



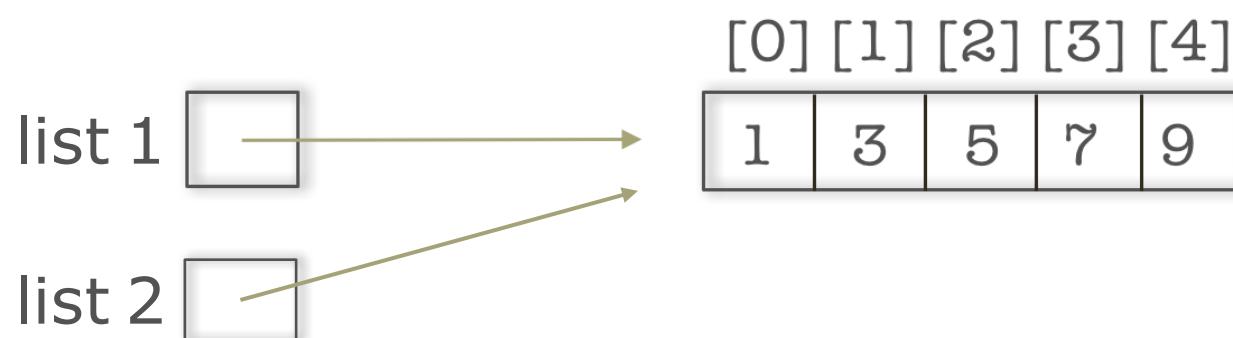
list 2 ?!

# Reference Semantics

```
int[] list1 = new int[5];
```

```
for (int i = 0; i < list1.length; i++)
    list1[i] = 2 * i + 1;
```

```
int[] list2 = list1;
```



# Reference Semantics

```
public class Test {  
  
    public static void main(String[] args) {  
        int[] list1 = new int[5];  
  
        for (int i = 0; i < list1.length; i++) {  
            list1[i] = 2 * i + 1;  
        }  
  
        int[] list2 = new int[5];  
        list2 = list1;  
  
        list2[0] = 10;  
        System.out.println(list1[0]);  
    }  
}
```

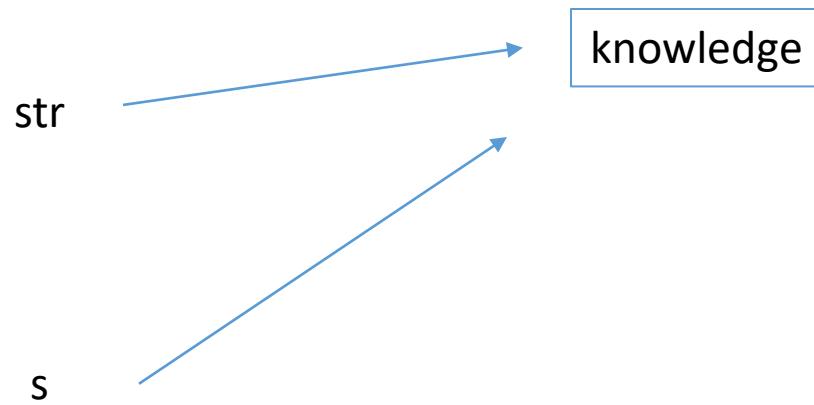
10

# String

- String is a reference type
- for which operator overloading is allowed (+ and +=)
- String objects are immutable
  - The value cannot be changed

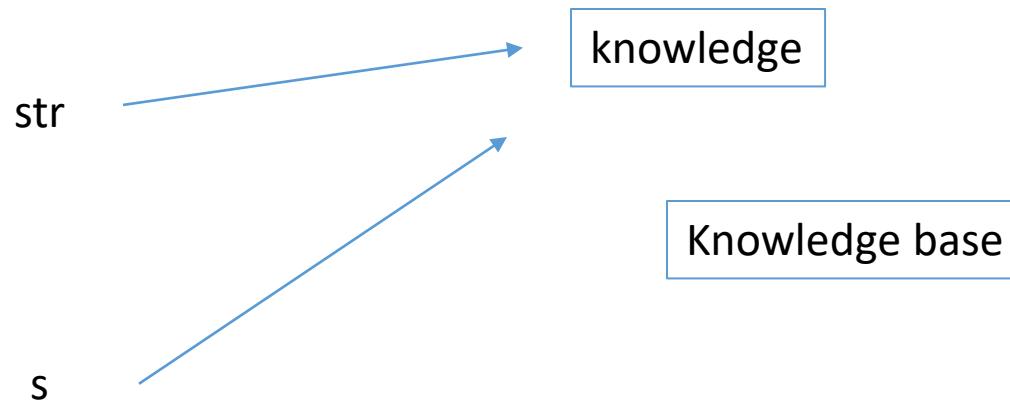
```
public class Test {  
  
    public static void main(String[] args) {  
        String str = "knowledge";  
        String s = str;  
        str = str.concat(" base");  
        System.out.println(str);  
        System.out.println(s);  
    }  
}
```

knowledge base



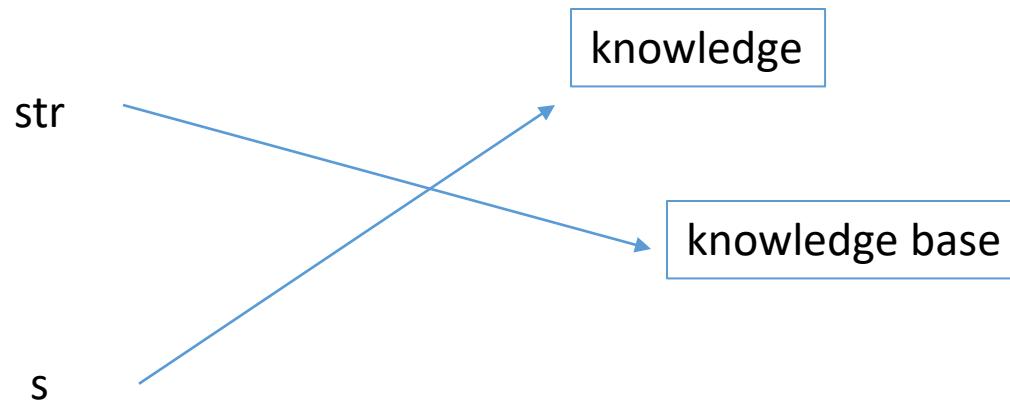
```
public class Test {  
  
    public static void main(String[] args) {  
        String str = "knowledge";  
        String s = str;  
        str = str.concat(" base");  
        System.out.println(str);  
        System.out.println(s);  
    }  
}
```

knowledge base



```
public class Test {  
  
    public static void main(String[] args) {  
        String str = "knowledge";  
        String s = str;  
        str = str.concat(" base");  
        System.out.println(str);  
        System.out.println(s);  
    }  
}
```

knowledge base



# String

- String is a reference type
- for which operator overloading is allowed (+ and +=)
- String is a series of chars

```
String string1 = "abc";
```

```
String string2 = string1;
```

```
System.out.println(string2);
```

abc

# String

- To compare string objects use equals and compareTo methods – not ==, !=, <, or >
  - Why?
- Other useful string methods:
  - length, charAt, substring
- Check Java online to see all methods

# Null

- A Java keyword signifying nothing
- this concept is only for reference types

```
String string1 = null;
```

```
String string2 = "";
```

# Exceptions

# Exceptions

- An exception is a special event that interrupts the control of the program
- Exceptions are “thrown” explicitly by the code

# Exceptions

- An exception is a special event that interrupts the control of the program
- Exceptions are “thrown” explicitly by the code
- Use a `try` block to wrap any code that might `throw` an exception
- A `catch` block immediately follows a `try` block

# Exceptions

- An exception is a special even that interrupts the control of the program
- Exceptions are “thrown” explicitly by the code
- Use a `try` block to wrap any code that might `throw` an exception
- A `catch` block immediately follows a `try` block
- Execution of the program jumps inside the `catch` block if an exception occurred within the `try` block

# Exceptions

```
try
{
    FileReader in = new FileReader("fakefile.txt");
}
catch(FileNotFoundException e)
{
    System.out.println("file does not exist");
}
```

Some code examples

```
public class OldMacDonaldPassing
{
    //-- the cow verse
    public static void cowVerse()
    {
        macDonald();
        had( "a cow" );
        with( "a Moo", "Moo" );
        macDonald();
    }
    //-- the pig verse
    public static void pigVerse()
    {
        macDonald();
        had( "a pig" );
        with( "an Oink", "Oink" );
        with( "a Moo", "Moo" );
        macDonald();
    }
    //-- the dog verse
    public static void dogVerse()
    {
        macDonald();
        had( "a dog" );
        with( "a Bow", "Wow" );
        with( "an Oink", "Oink" );
        with( "a Moo", "Moo" );
        macDonald();
    }
    //-- start and end of each verse
    public static void macDonald()
    {
        System.out.println( "Old MacDonald had a farm, E-I-E-I-O" );
    }
    //-- the "Had" line
    public static void had( String name )
    {
        System.out.println( "And on his farm he had " + name
            + ", E-I-E-I-O" );
    }
    //-- the "With a" lines
    public static void with( String xxx, String yyy )
    {
        System.out.println( "With " + xxx + ", " + yyy + " here" );
        System.out.println( "And " + xxx + ", " + yyy + " there" );
        System.out.println( "Here " + xxx + ", there " + xxx );
        System.out.println( "Everywhere " + xxx + ", " + yyy );
    }
    //-- main
    public static void main( String[] args )
    {
        cowVerse();
        System.out.println();
        pigVerse();
        System.out.println();
        dogVerse();
    }
}
```

```
import java.util.*;
public class BMIFixedHeight
{
    public static void main( String[] args )
    {
        Scanner keyboard = new Scanner( System.in );
        System.out.print( "Minimum weight, maximum weight, height: " );
        int minWeight = keyboard.nextInt();
        int maxWeight = keyboard.nextInt();
        double height = keyboard.nextDouble();
        int size = maxWeight - minWeight + 1;
        double[] bmi = new double[ size ];
        for ( int i = 0; i < size; i ++ )
        {
            bmi[ i ] = 703.0 * ( minWeight + i ) / height / height;
        }
        for ( int i = 0; i < size; i ++ )
        {
            System.out.printf( "%3d:%5.2f%n", ( minWeight + i ), bmi[ i ] );
        }
    }
}
```

```
public class ArraySwap
{
    public static void print( String[] words )
    {
        for ( int i = 0; i < words.length; i ++ )
        {
            System.out.print( " " + words[ i ] );
            if ( i % 8 == 7 || i == words.length - 1 )
            {
                System.out.println();
            }
        }
    }
    public static void main( String[] args )
    {
        String[] words = new String[]{
            "Several", "Species", "of", "Small", "Furry", "Animals",
            "Gathered", "Together", "in", "a", "Cave", "and",
            "Grooving", "With", "a", "Pict" };
        print( words );

        String temp = words[ 3 ];
        words[ 3 ] = words[ 5 ];
        print( words );

        words[ 5 ] = temp;
        print( words );
    }
}
```

```
import java.util.Scanner;
public class StringIndices
{
    public static void main( String[] args )
    {
        Scanner keyboard = new Scanner( System.in );
        System.out.print( "Enter string: " );
        String input = keyboard.nextLine();
        for ( int i = 0; i <= input.length(); i ++ )
        {
            System.out.print( "position = " + i );
            System.out.println( " .. char is " + input.charAt( i ) );
        }
    }
}
```

```
public class CharConversion
{
    public static void main( String[] args )
    {
        char c1 = 'a';
        char c2 = 'b';
        char c3 = (char)( c1 + 2 );
        boolean res = c1 > c2;
        int diff = 'Z' - 'A';
        System.out.println(
            "(c1>c2)=" + res + ", diff=" + diff + ", c3=" + c3 );
    }
}
```

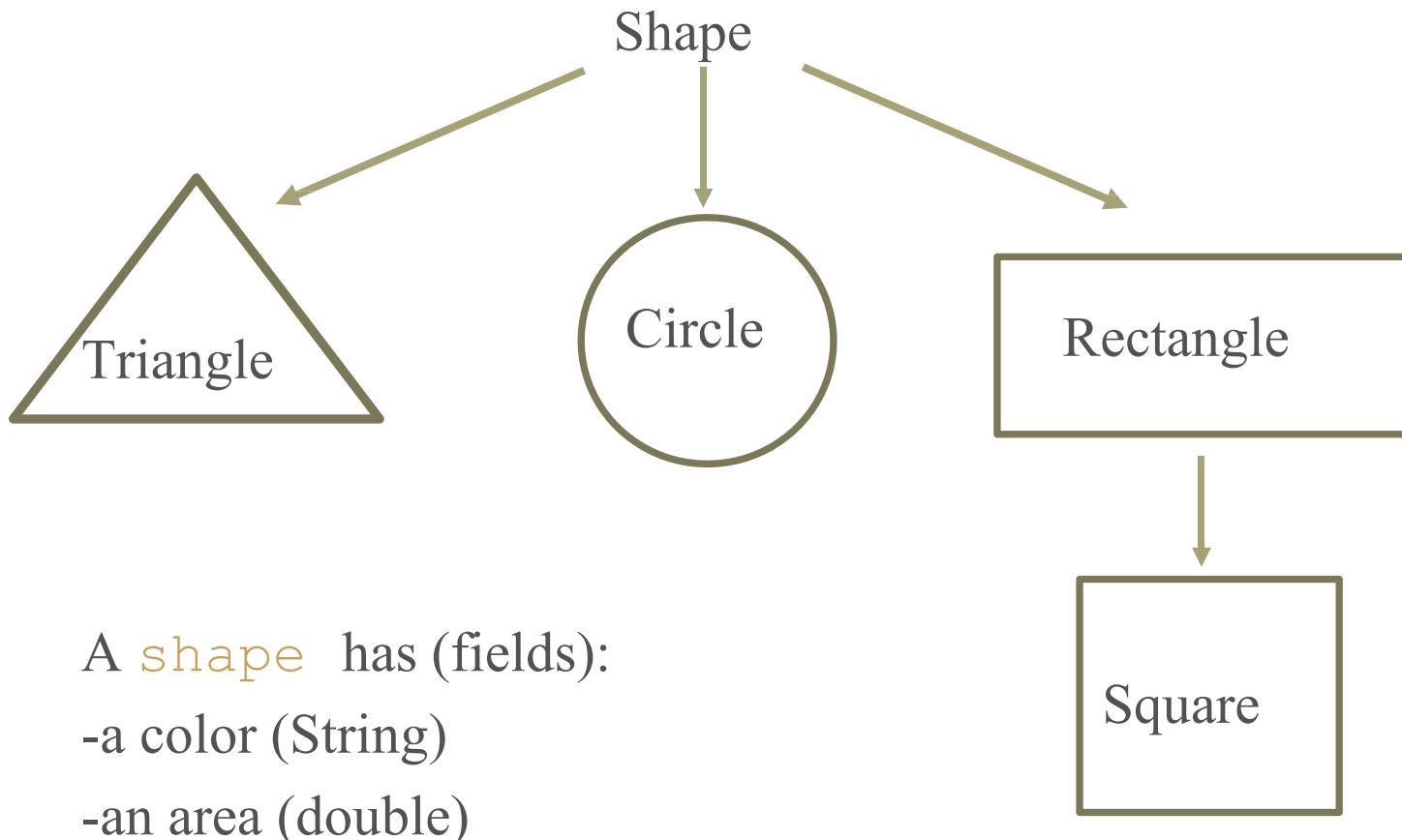
Next time...

- Reading
  - Chapter 8
- Lab
- Homework
  - Upload to BOX

# Inheritance, interfaces

## CSC220|Computer Programming 2

# Shape



```
public class Triangle{  
    String color;  
    double area;  
}  
  
public class Circle{  
    String color;  
    double area;  
}  
  
public class Rectangle{  
    String color;  
    double area;  
}  
  
public class Square{  
    String color;  
    double area;  
}
```

what if I want to  
redefine color as  
an integer array (R,G,B)?

What if I want to  
give each shape an  
outline color?

**Whatt can I  
do?  
extends**

```
public class Shape{  
    String color;  
    double area;  
}
```

```
public class Triangle extends Shape{  
}
```

```
public class Circle extends Shape{  
}
```

```
public class Rectangle extends Shape{  
}
```

```
public class Square extends Rectangle{  
}
```

```
public class Shape{  
    String color;  
    double area;  
}  
  
public class Triangle extends Shape{  
}  
  
public class Circle extends Shape{  
}  
  
public class Rectangle extends Shape{  
}  
  
public class Square extends Rectangle{  
}
```

Inherit all public  
fields and  
methods of  
Shape

```
public class Shape{  
    String color;  
    double area;  
}
```

Called the  
base class  
or  
superclass

```
public class Triangle extends Shape{  
}
```

```
public class Circle extends Shape{  
}
```

```
public class Rectangle extends Shape{  
}
```

```
public class Square extends Rectangle{  
}
```

Inherit all public  
fields and  
methods of  
Shape

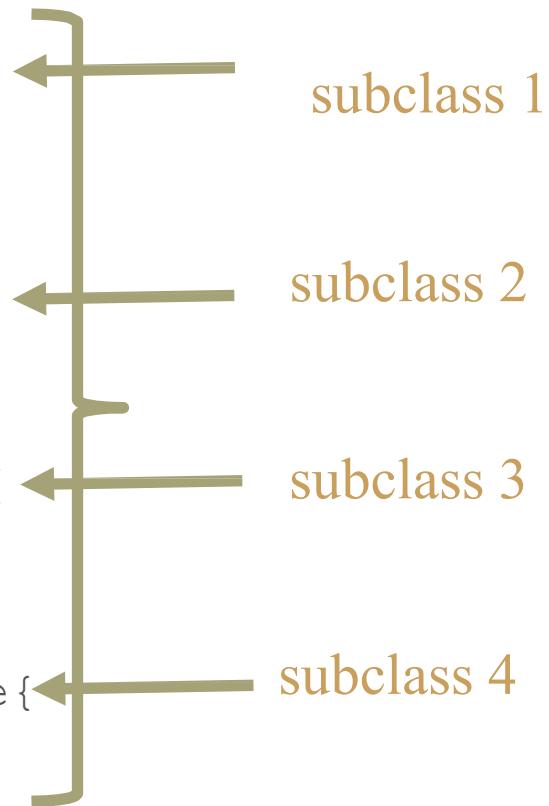
```
public class Shape{  
    String color;  
    double area;  
}
```

```
public class Triangle extends Shape{  
}
```

```
public class Circle extends Shape{  
}
```

```
public class Rectangle extends Shape{  
}
```

```
public class Square extends Rectangle{  
}
```



# Terminology

- Inheritance (inherit)
  - A programming technique that allows a derived class to extend the functionality of a base class, inheriting all of its state and behavior.
- Superclass
  - The parent class in an inheritance relationship: Shape
- Subclass
  - The child, or derived, class in an inheritance relationship: Triangle, Circle, Rectangle

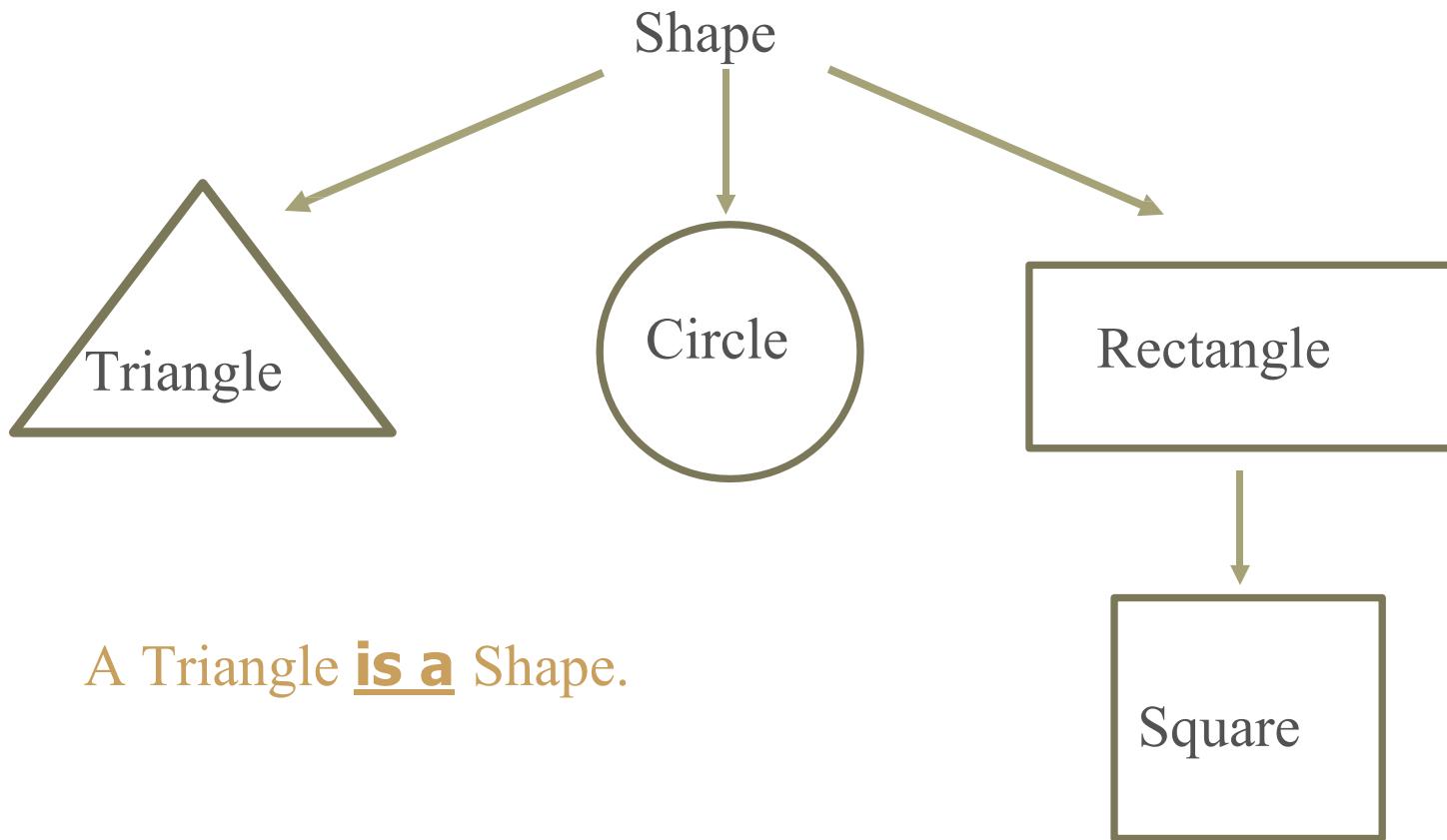
# extends

```
public class <name> extends <superclass> {  
    ...  
}
```

- Subclass **extends** the superclass  
because subclass

Receives the superclass's **state** and **behavior**

# Shape



# Single inheritance

- A java class can have only one superclass.
- One class may be extended by many subclasses.

- Now we have several shape classes, all with common fields associated with every shape
- But...
  - Circles have a radius
  - Rectangles have a width and height
  - Triangles have three Points

- Now we have several shape classes, all with common fields associated with every shape
- But...
  - Circles have a radius
  - Rectangles have a width and height
  - Triangles have three Points
- Does it make sense for all shapes to have a radius? a width and height? Or three Points?

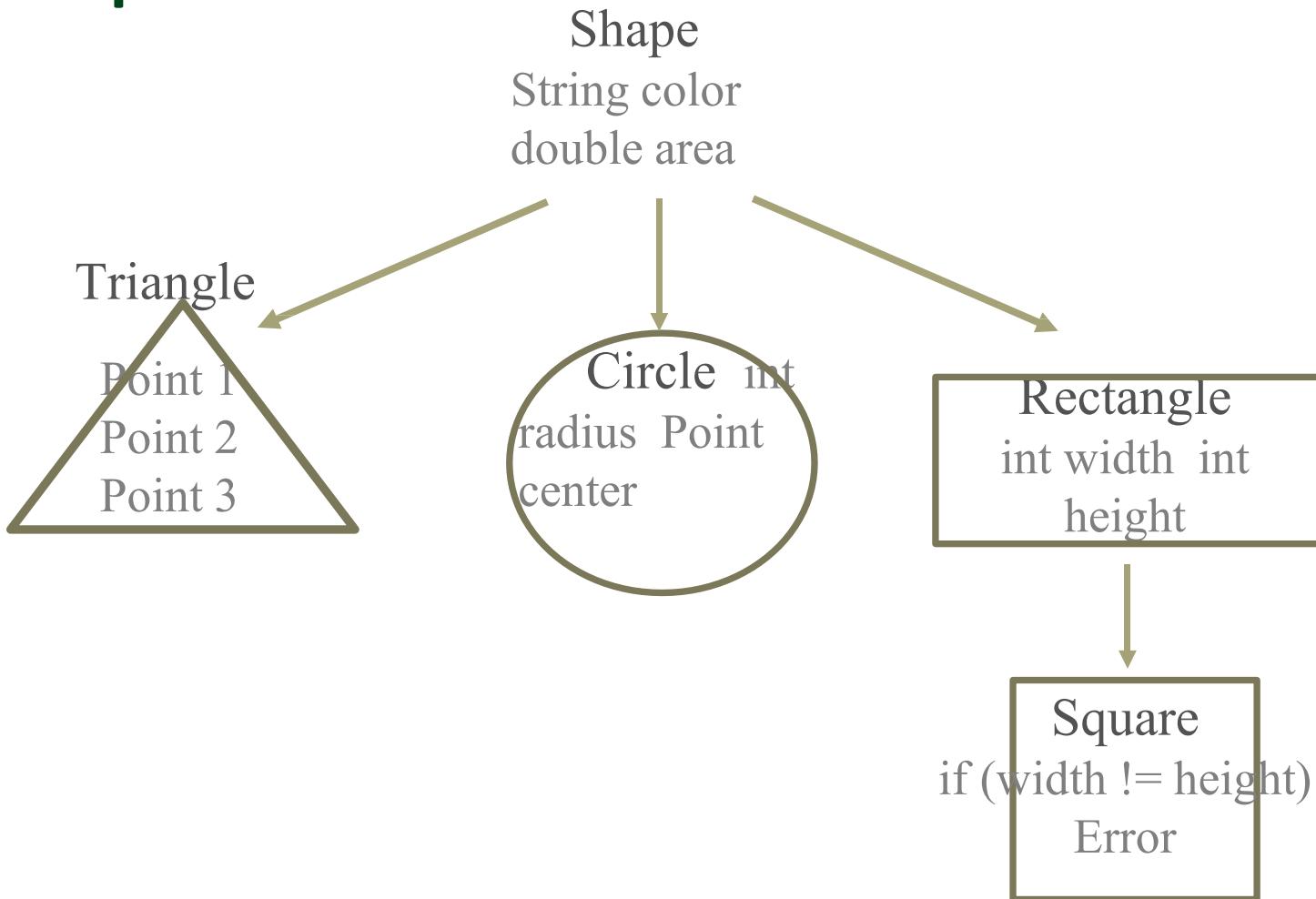
- Now we have several shape classes, all with common fields associated with every shape
- But...
  - Circles have a radius
  - Rectangles have a width and height
  - Triangles have three Points
- Does it make sense for all shapes to have a radius? a width and height? Or three Points?
- inherited classes can add their own fields and methods.

# extends

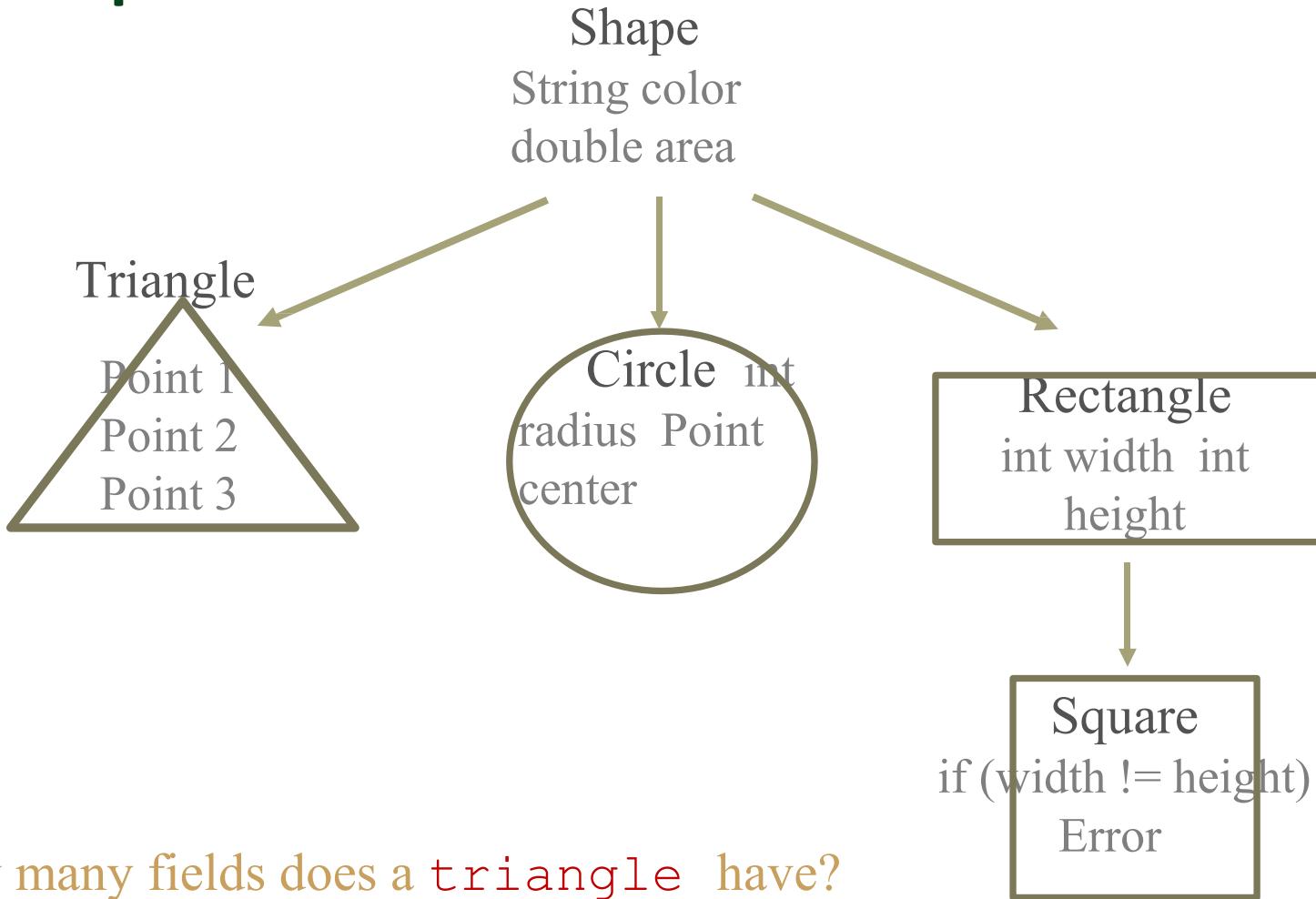
```
public class <name> extends <superclass> {  
    ...  
}
```

- Subclass `extends` the superclass because
  - Receives the superclass's state and behavior
  - Can also add new state and behavior of its own

# Shape

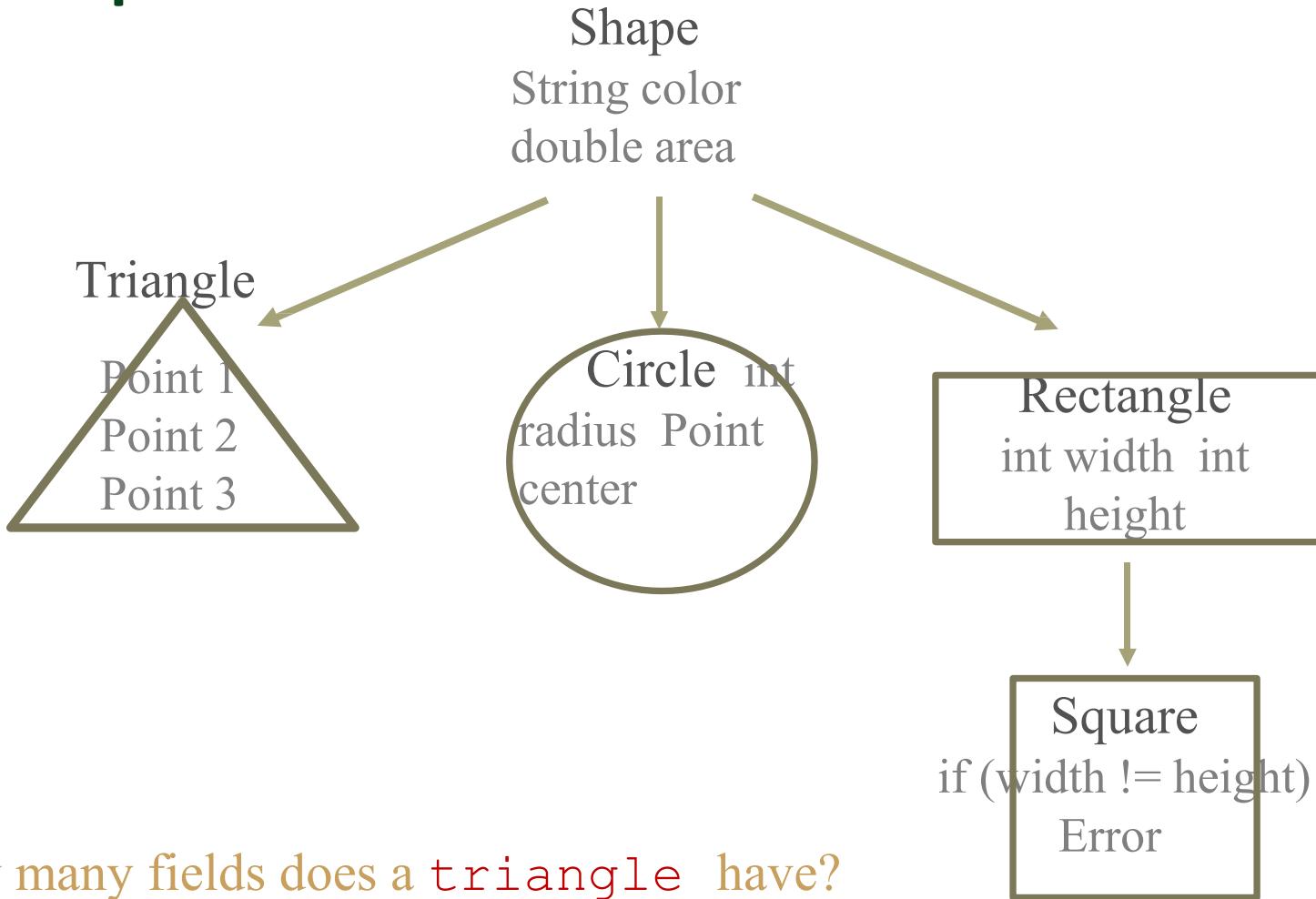


# Shape



How many fields does a triangle have?

# Shape



How many fields does a **triangle** have?

5

# Inherited classes also inherit methods

```
public class Shape {  
    String color;  
    double area;  
  
    public String toString(){  
        return color + " shape";  
    }  
}
```

---

```
public class Circle extends Shape {}
```

---

```
public static void main(String[] args){  
    Circle c = new Circle();  
    c.color = "red";  
    System.out.println(c.toString());  
}
```

# Inherited classes also inherit methods

```
public class Shape {  
    String color;  
    double area;  
  
    public String toString(){  
        return color + " shape";  
    }  
}
```

---

```
public class Circle extends Shape {}
```

---

```
public static void main(String[] args){  
    Circle c = new Circle();  
    c.color = "red";  
    System.out.println(c.toString());  
}
```

→ red shape

# What can('t) inherited classes do?

- A **derived** class **can**:
  - Add new fields
  - Add new methods
- A **derived** class **cannot**:
  - Remove fields
  - Remove methods
  - Inherit private fields
  - Inherit private methods

Derived class is another name for subclass

# Overriding methods

- To implement a new version of a method to replace code that would otherwise have been inherited from a superclass
- Replacing behavior from the superclass is called *overriding*.

# Overriding methods

- Method MUST have the same signature
  - Same name, parameters, return type*

```
public class Shape {  
    String color;  
    double area;  
  
    public String toString(){  
        return color + " shape";  
    }  
}  
  
public class Circle extends Shape {  
    int radius;  
    Point center;  
  
    // override  
    public String toString(){  
        return color + " circle with radius:" + radius;  
    }  
}
```

---

```
public static void main(String[] args){  
    Circle c = new Circle();  
    c.color = "red";  
    c.radius = 10;  
    System.out.println(c.toString());  
}
```

# Overriding methods

- Method MUST have the same signature
  - Same name, parameters, return type*

```
public class Shape {  
    String color;  
    double area;  
  
    public String toString(){  
        return color + " shape";  
    }  
}
```

```
public class Circle extends Shape {  
    int radius;  
    Point center;  
  
    // override  
    public String toString(){  
        return color + " circle with radius:" + radius;  
    }  
}
```

---

```
public static void main(String[] args){  
    Circle c = new Circle();  
    c.color = "red";  
    c.radius = 10;  
    System.out.println(c.toString()); →  
}  
red circle with radius:10
```

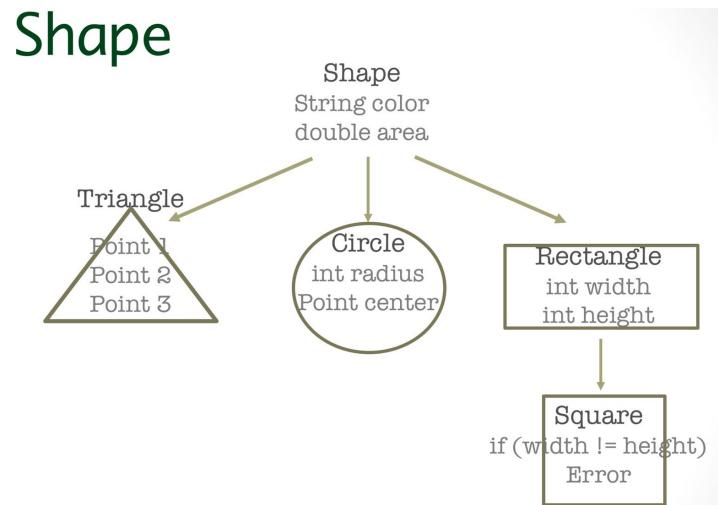
# Why override?

- There may be a method that makes sense for all shapes to have, but with drastically different implementations

# Why override?

- There may be a method that makes sense for all shapes to have, but with drastically different implementations

```
public double getArea () {  
    ...  
}
```



# Why override?

- There may be a method that makes sense for all shapes to have, but with drastically different implementations

```
public double getArea() {  
    ...  
}
```

Is the area computation the same for a  
Circle and a Square?

# Partial Overriding

- When we want to do something just slightly different than the base class, but most of the code is done for us in the superclass method
- How to explicitly call the superclass's version of a method

```
public void doSomething() {  
    super.doSomething();  
    // then do a little more  
}
```

# Partial override

```
public class Shape {  
    String color;  
    double area;  
  
    public String toString(){  
        return color;  
    }  
}
```

```
public class Circle extends Shape {  
    int radius;  
    Point center;  
  
    // override  
    public String toString(){  
        return super.toString() +  
            " circle with radius:" + radius;  
    }  
}
```

```
public static void main(String[] args){  
    Circle c = new Circle();  
    c.color = "red";  
    c.radius = 10;  
    System.out.println(c.toString());  
}
```

red circle with radius:10

# super

- Constructors are not inherited
- Subclasses have to have their own constructor
- A subclass's constructor must always begin by calling a constructor from the superclass.
  - Must be the first statement in the subclass's constructor
- How?
  - Use `super` keyword...

# super

```
public class Shape {  
    String color;  
    double area;  
  
    public Shape(){  
  
        public String toString(){  
            return color;  
        }  
    }  
}
```

```
public class Circle extends Shape {  
    int radius;  
    Point center;  
  
    public Circle(){  
        super();  
    }  
  
    public Circle(int radius){  
        super();  
        this.radius = radius;  
    }  
  
    // override  
    public String toString(){  
        return super.toString() +  
            " circle with radius:" + radius;  
    }  
}
```

# The Object class

- Ultimate superclass of all other Java classes
  - Even when not declared explicitly
- Object class contains methods that are common to all objects
  - `toString...`
- When to use?

# The Object class

- Ultimate superclass of all other Java classes
  - Even when not declared explicitly
- Object class contains methods that are common to all objects
  - `toString...`
- When to use?
  - Writing a method that can accept any object as a parameter

```
Public static void myMethod (Object o) {  
    ...  
}
```

# instanceof

- Tests whether a variable refers to an object of a given type
- ```
public boolean equals (Object o) {  
    if (o instanceof Point) {  
        Point other = (Point) o;  
        return (x == other.x && y == other.y);  
    } else {  
        return false;  
    }  
}
```

# Polymorphism

# Polymorphism

- Type compatibility
  - It is legal for a superclass variable to refer to an object of its subclass.

```
Shape c = new Circle();
```

# Polymorphism

- Type compatibility
  - It is legal for a superclass variable to refer to an object of its subclass.

```
Shape c = new Circle();
```

- No type conversion occurs: the object referred to by "c" is actually a Circle object.
- If a method called "c", it behaves like a Circle object.

# Type compatibility

```
public static boolean isLarger(Shape s1, Shape s2){  
    return s1.getArea() > s2.getArea();  
}  
  
public static void main(String[] args){  
  
    Rectangle r = new Rectangle();  
    Circle c = new Circle();  
  
    if (isLarger(r,c)){  
        System.out.println("Code is fine so far!");  
    }  
}
```

# Type compatibility

```
public static boolean isLarger(Shape s1, Shape s2){  
    return s1.getArea() > s2.getArea();  
}  
  
public static void main(String[] args){  
  
    Rectangle r = new Rectangle();  
    Circle c = new Circle();  
  
    if (isLarger(r, c)){  
        System.out.println("Code is fine so far!");  
    }  
}
```

why can I pass `isLarger` a `Circle` and a `Triangle`?

- polymorphism is a fancy word for automatically determining an object's type at runtime
- the most specific type possible is used

```
Shape s1 = new Circle();
```

```
Shape s2 = new Triangle();
```

```
s1.getArea();    What type is s1 treated as?
```

```
s2.getArea();
```

- polymorphism is a fancy word for automatically determining an object's type at runtime
- the most specific type possible is used

```
Shape s1 = new Circle();
```

```
Shape s2 = new Triangle();
```

s1.getArea();      What type is s1 treated as?

s2.getArea();      What type is s2 treated as?

- polymorphism is a fancy word for automatically determining an object's type at runtime
- the most specific type possible is used

```
Shape s1 = new Circle();
```

```
Shape s2 = new Triangle();
```

- suppose Triangle does not override

```
toString() s2.toString();
```

What type is s2 treated as?

# Why polymorphism

- Lets us pass many different types of objects as parameters (in this case, Shapes).
- The code will produce different behavior depending on the type that is passed.

```
Shape[] shape_array = new Shape[3];
shape_array[0] = new Triangle();
shape_array[1] = new Circle();
shape_array[2] = new Rectangle();

//find the total area of all the shapes
double total_area = 0;
for(int i=0; i<3; i++)
    total_area += shape_array[i].getArea();

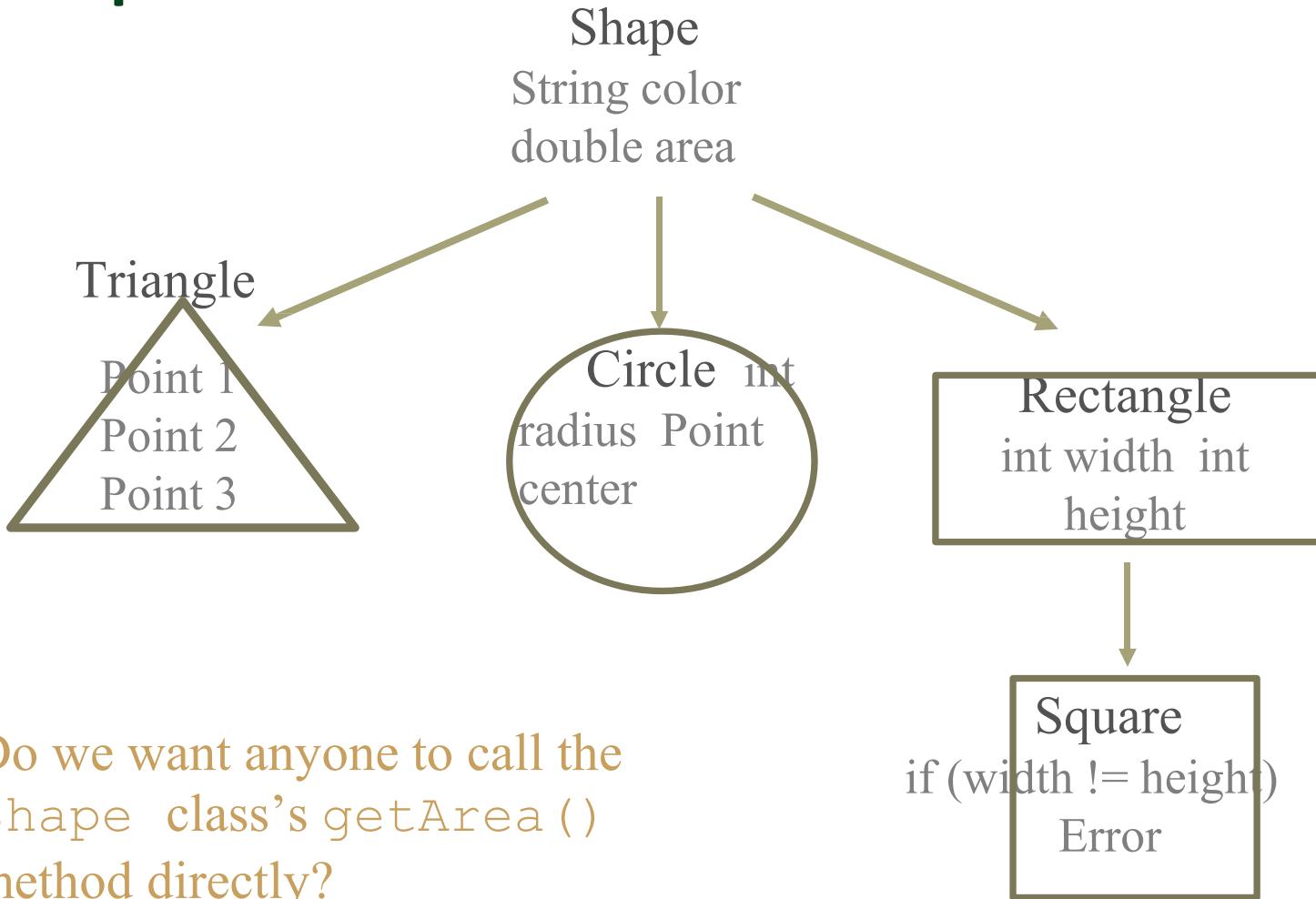
System.out.println(total_area);
```

Is this code valid?

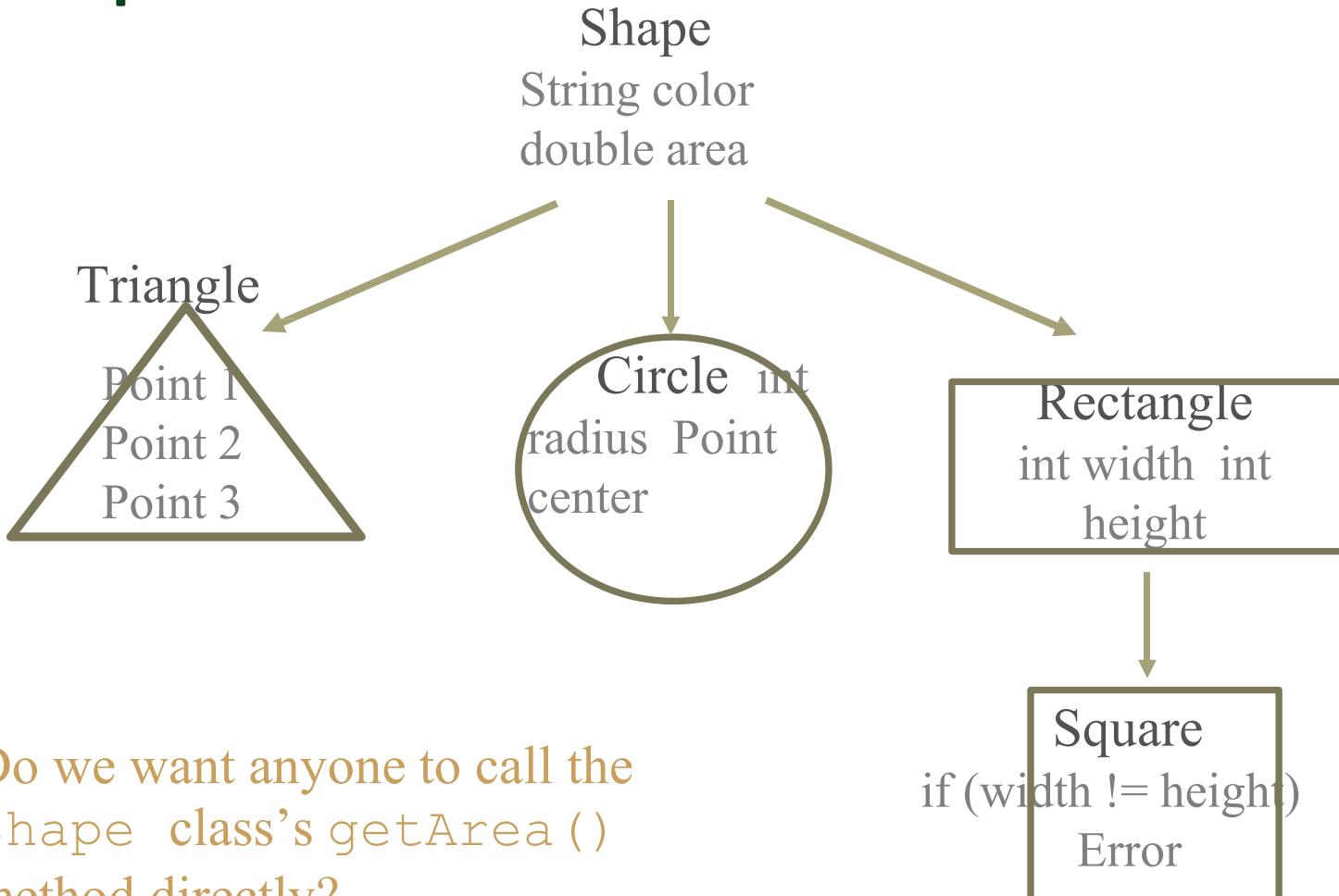
Is polymorphism happening?

# Interfaces

# Shape

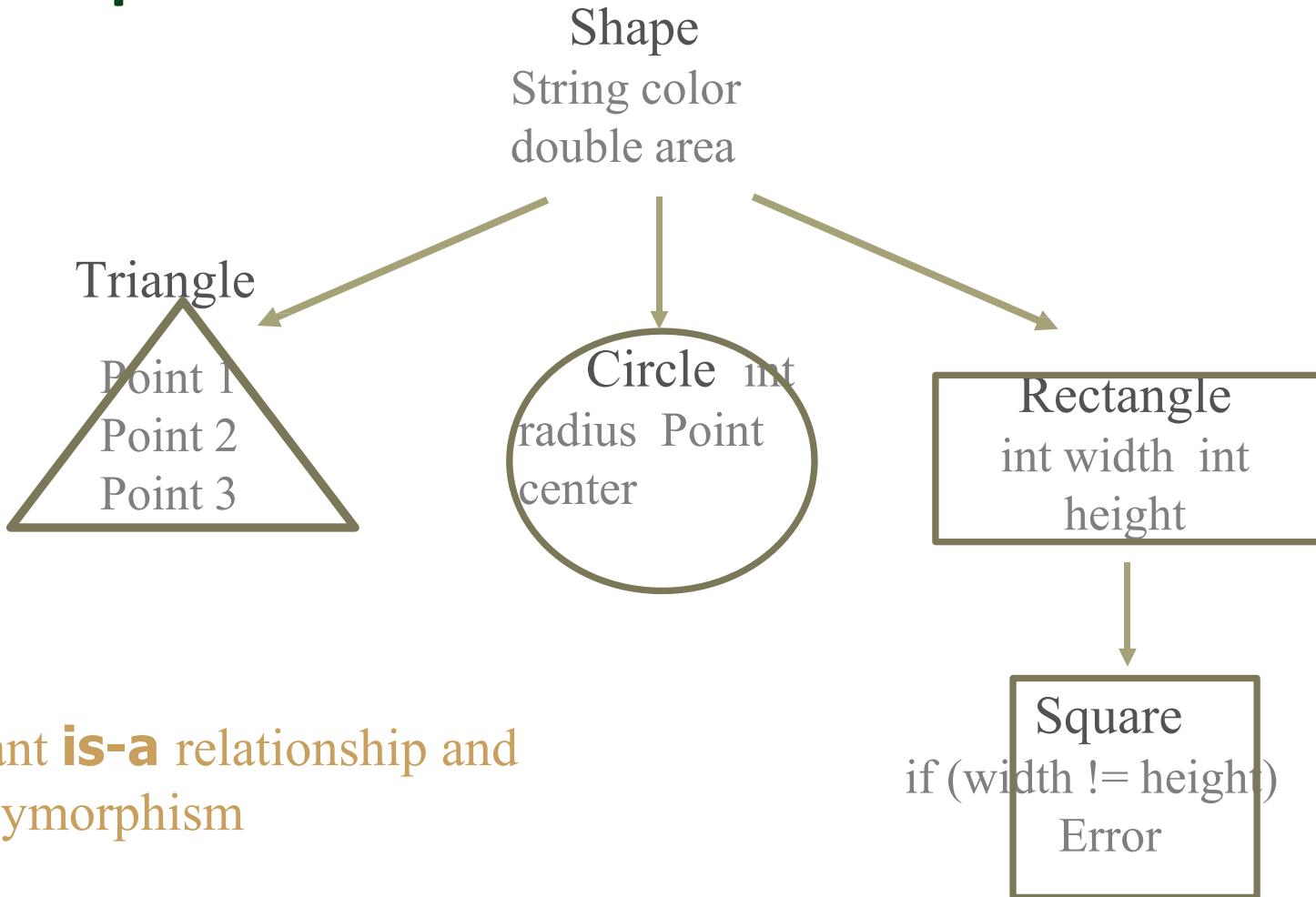


# Shape

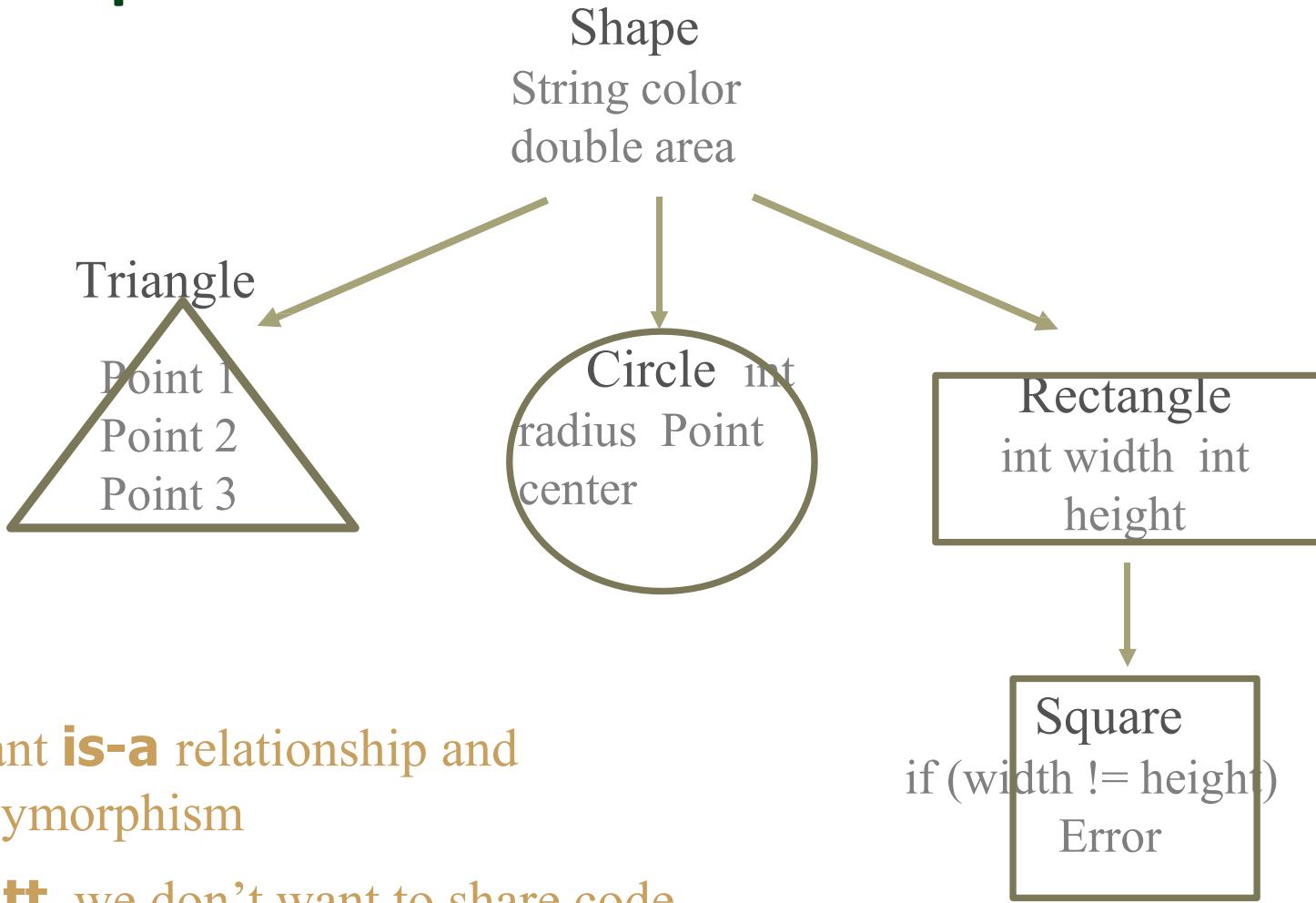


→ meant to be called from a specific shape

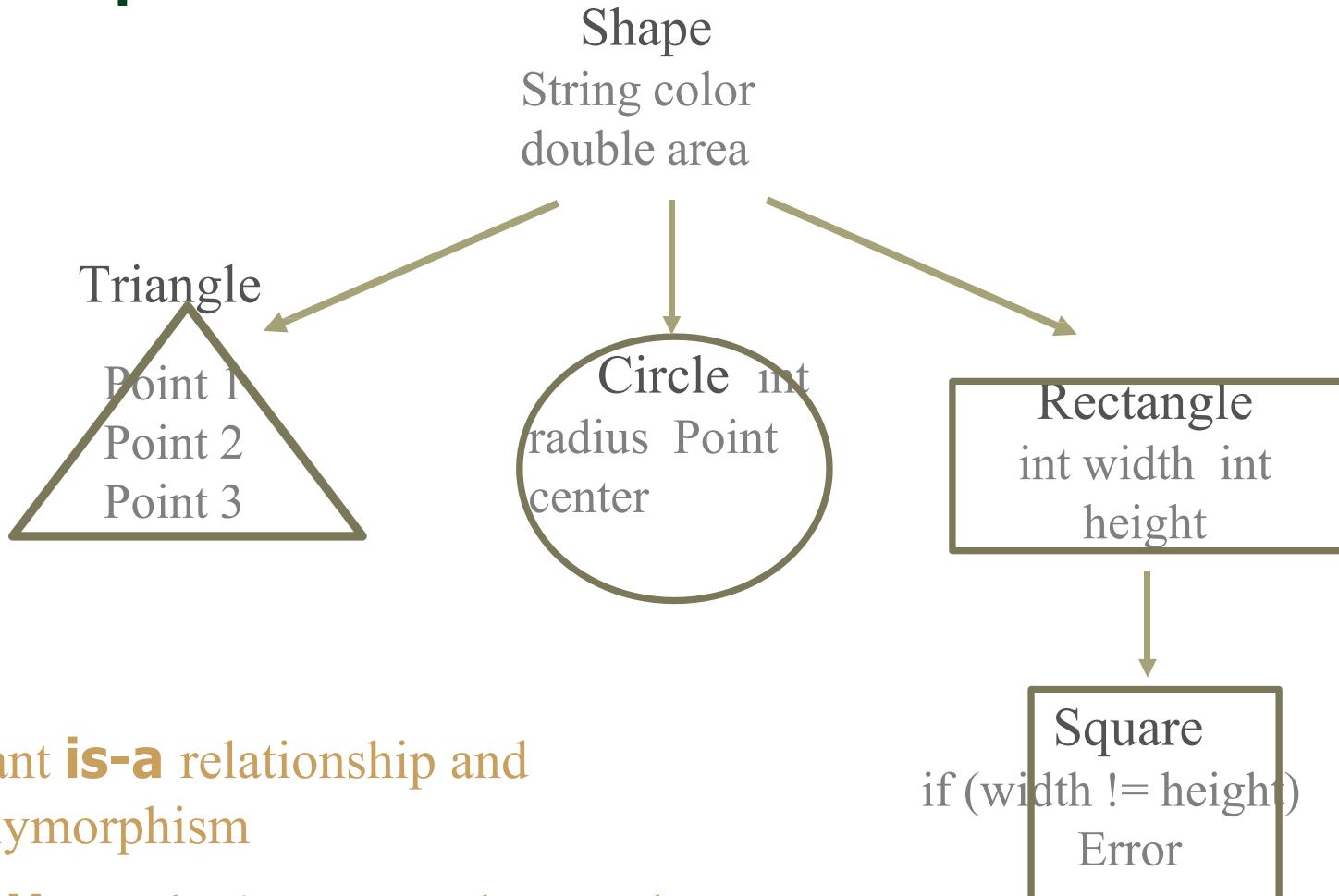
# Shape



# Shape



# Shape



Want **is-a** relationship and polymorphism

**Butt**, we don't want to share code

Inheritance doesn't work that way!

# interface

A Java feature to define a polymorphic hierarchy of classes without sharing code between them.

# interface

- A type that consists of a set of method declarations; Like a class but
  - It contains only method headers without bodies

# interface

- A type that consists of a set of method declarations; Like a class but
  - It contains only method headers without bodies
  - Class(es) promise to ***implement*** an interface
    - All the methods declared in the interface

# interface

- A type that consists of a set of method declarations; Like a class but
  - It contains only method headers without bodies
  - Class(es) promise to ***implement*** an interface
    - All the methods declared in the interface
    - Classes that implement an interface form an **is-a** relationship with that interface.

# interface

- A type that consists of a set of method declarations; Like a class but
  - It contains only method headers without bodies
- Class(es) promise to **implement** an interface
  - All the methods declared in the interface
- Classes that implement an interface form an **is-a** relationship with that interface.
- Polymorphism can handle objects from any of the classes that implement an interface.

# An interface for Shapes

- Represents the common functionality of all shapes

```
Public interface Shape{  
    ...  
}
```

# An interface for Shapes

- Represents the common functionality of all shapes

```
Public interface Shape{  
    ...  
}
```



Write headers for  
Shape methods that we  
want a Shape to  
contain.

# An interface for Shapes

```
public interface Shape{  
    public double getArea();  
    public double getPerimeter();  
}
```

- We don't specify how the methods are implemented.
- Any class that want to be considered a shape **MUST** implement these methods.
- Illegal for an interface to contain method bodies.
  - Only contain method headers
  - Class constants

# An interface for Shapes

```
public interface Shape{  
    public double getArea(); Abstract method  
    public double getPerimeter();  
}
```

- abstract method: is declared (in an interface) but not implemented.
  - Represent the behavior that a class promises to implement when it implements an interface.

# An interface for Shapes

```
public interface Shape{  
    public double getArea(); Abstract method  
    public double getPerimeter();  
}
```

- abstract method: is declared (in an interface) but not implemented.
  - Represent the behavior that a class promises to implement when it **implements** an interface.
- An interface **cannot** be instantiated.  
Shape p1; —————→ valid  
p1 = new Shape(); —————→ Error!

# Implementing Shape interface

```
Public class <name> implements <interface>{  
    ...  
}
```

# Implementing Shape interface

```
public class Rectangle implements Shape {  
    private double width;  
    private double height;  
  
    public Rectangle(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
  
    public double getArea(){  
        return width*height;  
    }  
  
    public double getPerimeter(){  
        return 2.0*(width + height);  
    }  
}
```

# Implementing Shape interface

```
public class Rectangle implements Shape {  
    private double width;  
    private double height;  
  
    public Rectangle(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
  
    public double getArea(){  
        return width*height;  
    }  
  
    public double getPerimeter(){  
        return 2.0*(width + height);  
    }  
}
```

# Benefits

- We can still achieve polymorphism

```
Shape[] shape_array = new Shape[3];
shape_array[0] = new Triangle();
shape_array[1] = new Circle();
shape_array[2] = new Rectangle();

//find the total area of all the shapes
double total_area = 0;
for(int i=0; i<3; i++)
    total_area += shape_array[i].getArea();

System.out.println(total_area);
```

# Benefits

- We can still achieve polymorphism
- Work around single inheritance
  - A class may extend only one superclass
  - A class may implement arbitrarily many interfaces.

```
Public class <name> extends <superclass>
    implements <interface>, ..., <interface>{
    ...
}
```

Next  
Time...

- quiz on Thursday
  - From Inheritance **and OOP**
  - You have a lab tomorrow
- Get a new assignment tomorrow
- Generic programming

# More Inheritance/Interface

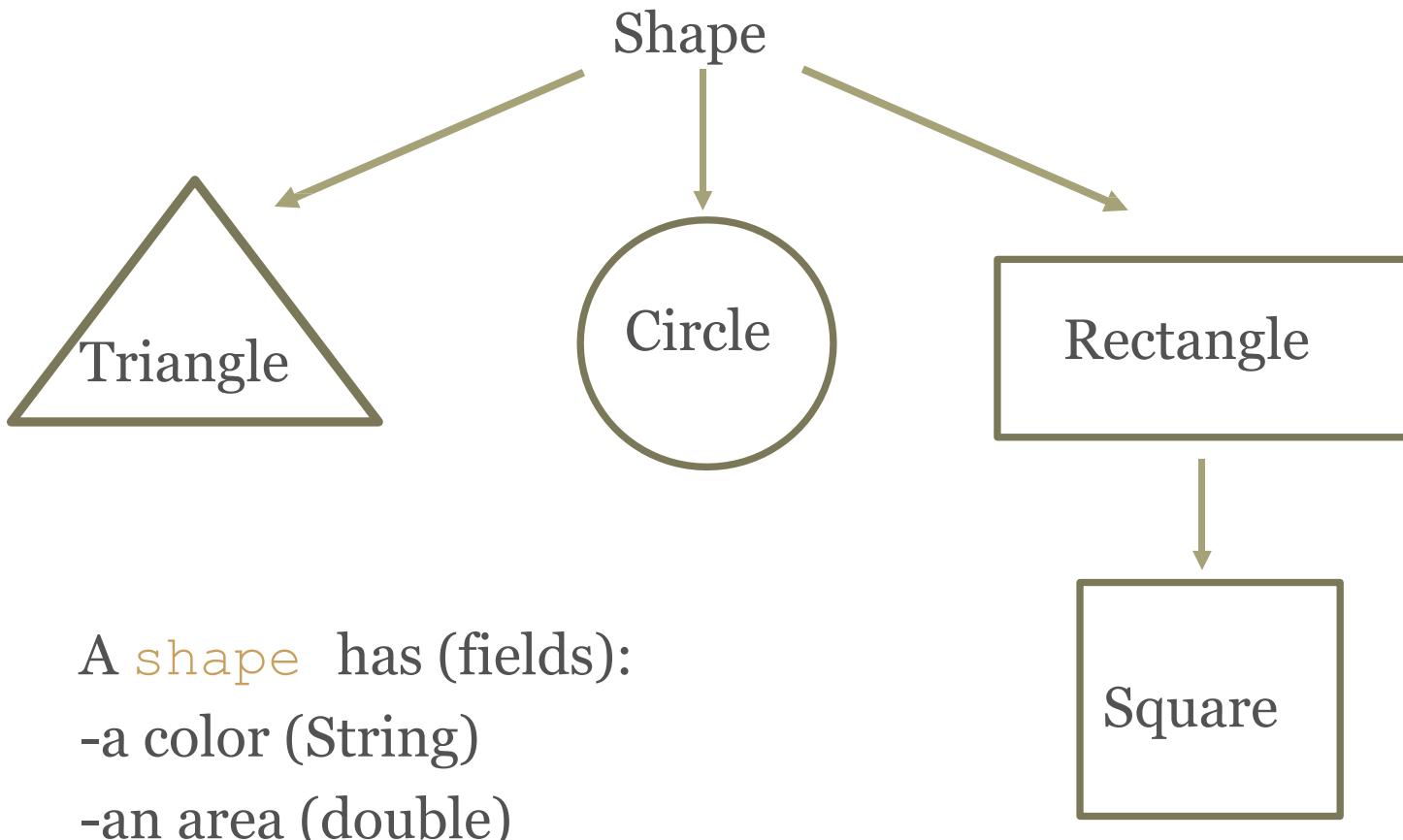
## CSC220 | Computer Programming 2

Last Time...

# Inheritance and interfaces

- A feature of OOP
- Reusability
  - Allow one class to be an extension of another
- Code on scale
  - Write programs with hierarchies of related object types.

# Shape



```
public class Shape{  
    String color;  
    double area;  
}
```

Called the  
base class  
or  
superclass

```
public class Triangle extends Shape{  
}
```

```
public class Circle extends Shape{  
}
```

```
public class Rectangle extends Shape{  
}
```

```
public class Square extends Rectangle{  
}
```

Inherit all  
public fields  
and methods of  
Shape

# What can('t) inherited classes do?

- A **derived** class **can**:
  - Add new fields
  - Add new methods
- A **derived** class **cannot**:
  - Remove fields
  - Remove methods
  - Inherit private fields
  - Inherit private methods

Derived class is another name for subclass

# Type compatibility & polymorphism

```
public static boolean isLarger(Shape s1, Shape s2){  
    return s1.getArea() > s2.getArea();  
}
```

```
public static void main(String[] args){  
  
    Rectangle r = new Rectangle();  
    Circle c = new Circle();  
  
    if (isLarger(r, c)){  
        System.out.println("Code is fine so far!");  
    }  
}
```

why can I pass `isLarger` a `Circle` and a `Triangle`?

# EXAMPLE

```
public class Employee {  
    public int getHours() {  
        return 40;  
    }  
  
    public double getSalary() {  
        return 40000.0;  
    }  
  
    public int getVacationDays() {  
        return 10;  
    }  
  
    public String getVacationForm() {  
        return "yellow";  
    }  
}
```

```
public class Lawyer extends Employee {
    public int getVacationDays() {
        return 15;
    }

    public String getVacationForm() {
        return "pink";
    }
}

public class Secretary extends Employee{
    public void takeDictation(String text) {
        System.out.println("Dictation text: " + text);
    }
}

public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 45000.0;
    }

    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }
}
```

```
public class EmployeeTester {  
    public static void main(String[] args) {  
        Employee edna = new Employee();  
        Lawyer lucy = new Lawyer();  
        Secretary stan = new Secretary();  
        LegalSecretary leo = new LegalSecretary();  
  
        printInfo(edna);  
        printInfo(lucy);  
        printInfo(stan);  
        printInfo(leo);  
    }  
  
    public static void printInfo(Employee e) {  
        System.out.print(e.getHours() + ", ");  
        System.out.printf("%.2f, ", e.getSalary());  
        System.out.print(e.getVacationDays() + ", ");  
        System.out.print(e.getVacationForm() + ", ");  
        System.out.println(e);  
    }  
}
```

40, \$40000.00, 10, yellow, Employee@10b30a7  
40, \$40000.00, 15, pink, Lawyer@1a758cb  
40, \$40000.00, 10, yellow, Secretary@1b67f74  
40, \$45000.00, 10, yellow, LegalSecretary@69b332

# EXAMPLE

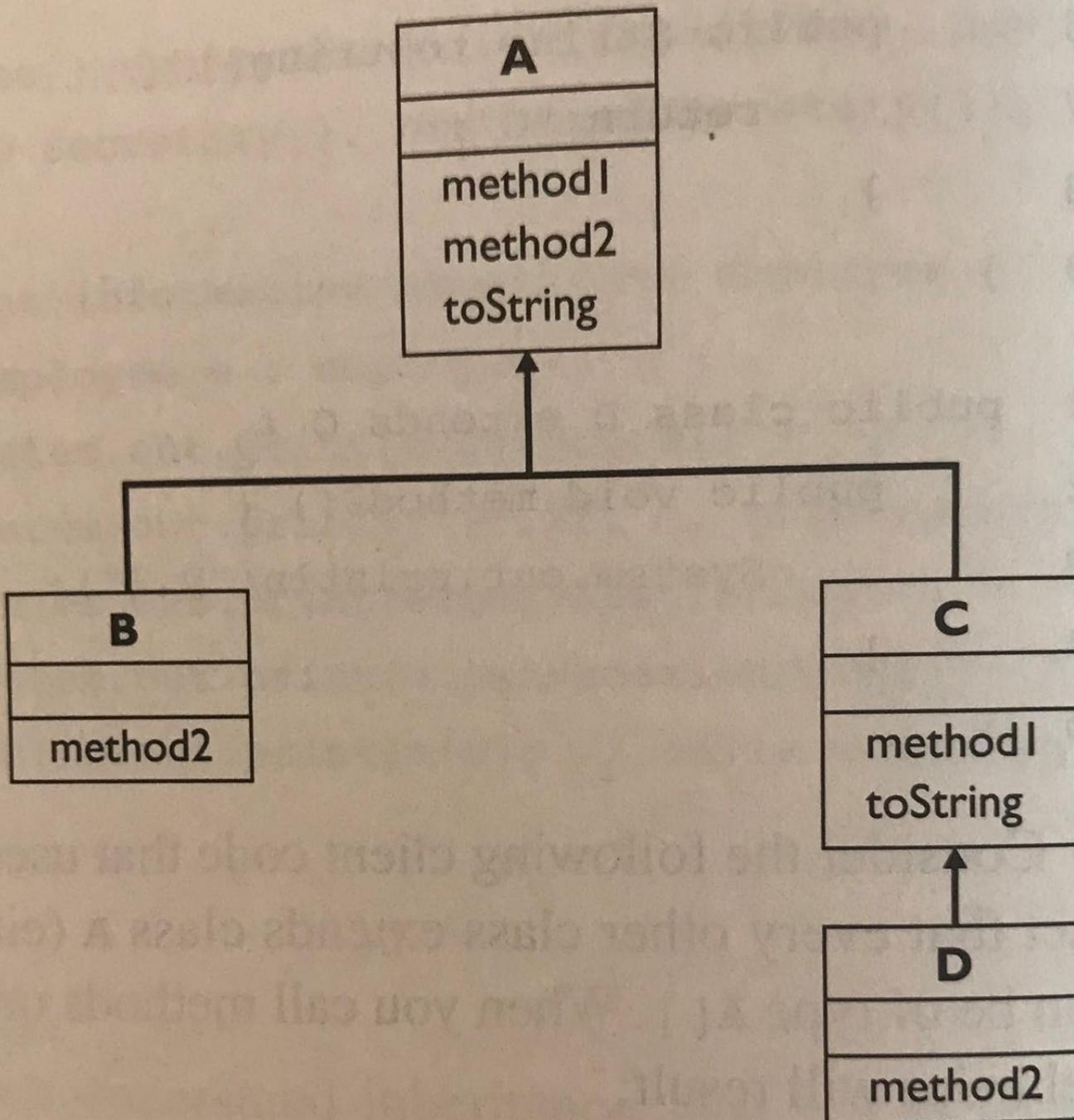
```
1 public class A {  
2     public void method1() {  
3         System.out.println("A 1");  
4     }  
5  
6     public void method2() {  
7         System.out.println("A 2");  
8     }  
9  
10    public String toString() {  
11        return "A";  
12    }  
13 }
```

```
1 public class B extends A {  
2     public void method2() {  
3         System.out.println("B 2");  
4     }  
5 }
```

```
1 public class C extends A {  
2     public void method1() {  
3         System.out.println("C 1");  
4     }  
5 }
```

```
6     public String toString() {  
7         return "C";  
8     }  
9 }
```

```
1 public class D extends C {  
2     public void method2() {  
3         System.out.println("D 2");  
4     }  
5 }
```



```
1 // Client program to use the A, B, C, and D classes.  
2 public class ABCDMain {  
3     public static void main(String[] args) {  
4         A[] elements = {new A(), new B(), new C(), new D()};  
5  
6         for (int i = 0; i < elements.length; i++) {  
7             System.out.println(elements[i]);  
8             elements[i].method1();  
9             elements[i].method2();  
10            System.out.println();  
11        }  
12    }  
13 }
```

```
1 public class E extends F {
2     public void method2() {
3         System.out.print("E 2 ");
4         method1();
5     }
6 }
1 public class F extends G {
2     public String toString() {
3         return "F";
4     }
5
6
7     public void method2() {
8         System.out.print("F 2 ");
9         super.method2();
10    }
11 }
1 public class G {
2     public String toString() {
3         return "G";
4     }
5
6
7     public void method1() {
8         System.out.print("G 1 ");
9     }
10
11     public void method2() {
12         System.out.print("G 2 ");
13     }
14 }
1 public class H extends E {
2     public void method1() {
3         System.out.print("H 1 ");
4     }
5
6 }
```

# interface

- A type that consists of a set of method declarations; Like a class but
  - It contains only method headers without bodies
- Class(es) promise to **implement** an interface
  - All the methods declared in the interface
- Classes that implement an interface form an **is-a** relationship with that interface.
- Polymorphism can handle objects from any of the classes that implement an interface.

# An interface for Shapes

```
public interface Shape{  
    public double getArea(); Abstract method  
    public double getPerimeter();  
}
```

- abstract method: is declared (in an interface) but not implemented.
  - Represent the behavior that a class promises to implement when it implements an interface.

# Implementing Shape interface

```
public class Rectangle implements Shape {  
    private double width;  
    private double height;  
  
    public Rectangle(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
  
    public double getArea(){  
        return width*height;  
    }  
  
    public double getPerimeter(){  
        return 2.0*(width + height);  
    }  
}
```

# Benefits

- We can still achieve polymorphism
- Work around single inheritance
  - A class may extend only one superclass
  - A class may implement arbitrarily many interfaces.

```
Public class <name> extends <superclass>
    implements <interface>, ..., <interface>{
    ...
}
```

Today...

# Generic Programming

- Suppose we want a data structure that just contains “things”
- We want it to:

- Suppose we want a data structure that just contains “things”
- We want it to:
  - Automatically grow if it gets full
  - Be able to remove items from it
  - Be able to add items to it

- Suppose we want a data structure that just contains “things”
- We want it to:
  - Automatically grow if it gets full
  - Be able to remove items from it
  - Be able to add items to it
- Will an array work?

```
Shape [ ] shape_array = new Shape [ 5 ] ;
```

- How about an ArrayList?

- How about an ThingsHolder?

```
public class ThingsHolder{  
    Shape storage[];  
    int capacity, numItems;  
  
    public void addItem(Shape item)  
    { /*some code*/ }  
  
    public void autoGrow()  
    { /*some code*/ }  
}
```

- How about an ThingsHolder?

```
public class ThingsHolder{  
    Shape storage[];  
    int capacity, numItems;  
  
    public void addItem(Shape item)  
    { /*some code*/ }  
  
    public void autoGrow()  
    { /*some code*/ }  
}
```

What is the problem with this?

# ArrayList

- The `ArrayList` class (from the `Collections` library) mimics an array and allows for dynamic expansion
- The `get, set` methods are used in place of `[]` for indexing
- the `add` method increases the size by one and adds a new item
- the `remove` method preserves the order of the list by shifting values to the left to fill in any gap

# ArrayList

ArrayList<E>

```
ArrayList<String> list = new ArrayList<String>();
```

type

type

```
Point p = new Point();
```

type

type

- This is why we always see <> associated with ArrayList

```
ArrayList<Shape> list = new ArrayList<Shape>();
```

- ArrayList is a **generic class** — we can create any version of it that we want
- **Generic programming:** algorithms are written in terms of types to-be-specified-later
  - Algorithms instantiated when needed for specific types defined by parameters

- Here's what the code actually looks like:

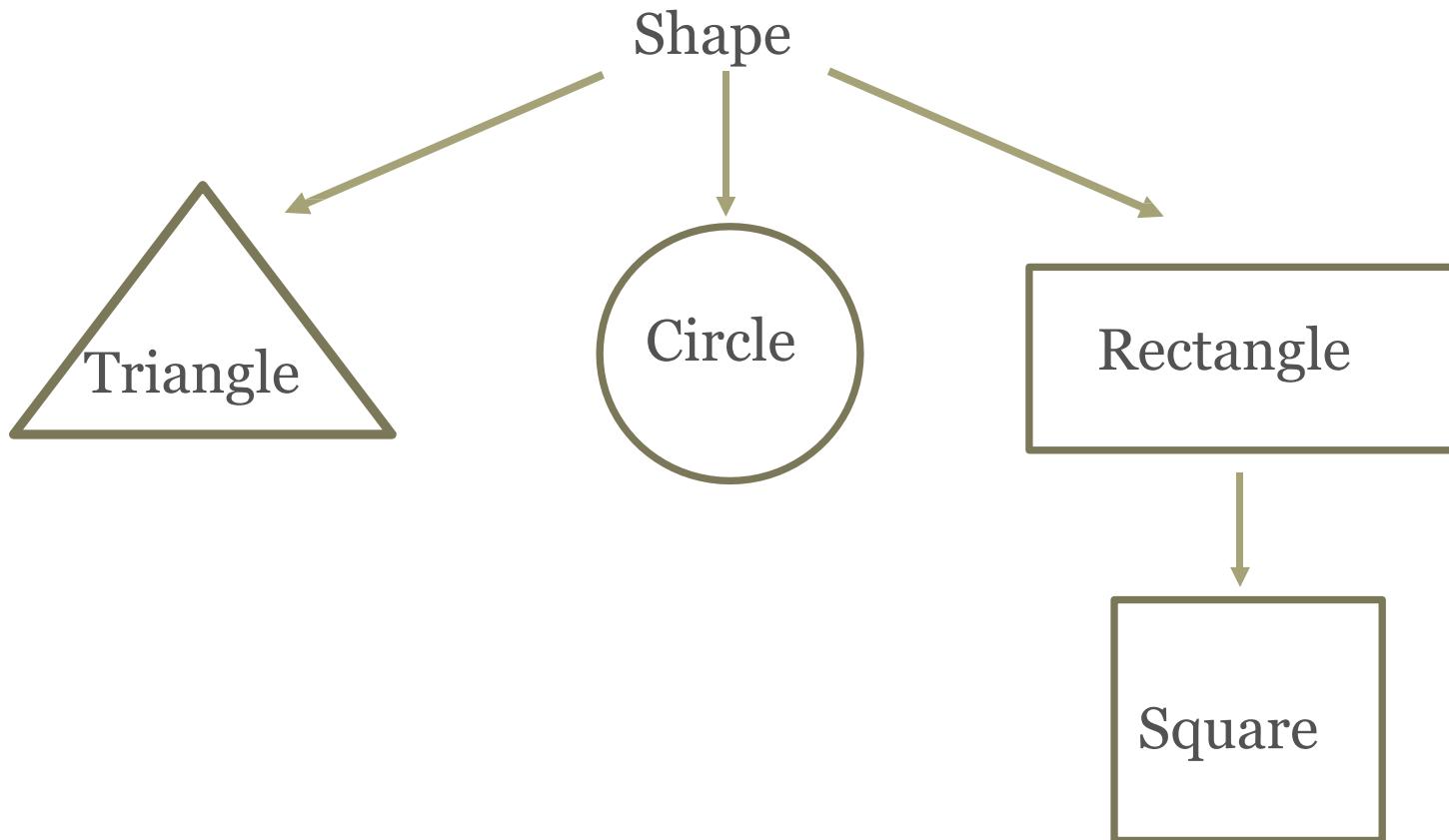
```
public class ThingsHolder<T> {
    T storage[];
    int capacity, numItems;

    public void add(T item)
    { ... }

    public T get(int i)
    { ... }
}
```

- The placeholder **T** is replaced with the real type when you instantiate an `ThingsHolder` with **<>**
- **T** can be used as a type anywhere in `ThingsHolder` class

# Inheritance hierarchy



# Generic placeholder <>

- The generic placeholder type is **VERY** specific
- `ArrayList<Triangle>` is not an `ArrayList<Shape>`, even though Triangle is a Shape!
- `ArrayList<type>` is only EXACTLY an `ArrayList<type>`, regardless of type's heritage

# Inheritance and generics

```
public void doStuff(ArrayList<Shape> a)  
{ . . . }
```

```
ArrayList<Triangle> tri_list;
```

```
ArrayList<Shape> shape_list;
```

```
doStuff(tri_list);
```



LEGAL?

```
doStuff(shape_list);
```



LEGAL?

# Inheritance and generics

```
public void doStuff(ArrayList<Shape> a)  
{ . . . }
```

```
ArrayList<Triangle> tri_list;
```

```
ArrayList<Shape> shape_list;
```

```
doStuff(tri_list);
```



ILLEGAL

```
doStuff(shape_list);
```



OK

# Inheritance and generics

```
public void doStuff(ArrayList<Shape> a)  
{ ... }
```

```
ArrayList<Triangle> tri_list;  
ArrayList<Shape> shape_list;
```

```
doStuff(tri_list); //ILLEGAL  
doStuff(shape_list); //OK
```

- We can still add Triangles to shape\_list
- Restriction applies only to the generic object itself

# Why generics?

- Everything in Java is an Object
- So, why not just make all data structures hold Objects?

# Why generics?

- Everything in Java is an Object
- So, why not just make all data structures hold Objects?
- Generics allow for type-checking at compile time instead of run-time

# Why generics?

- Everything in Java is an Object
- So, why not just make all data structures hold Objects?
- Generics allow for type-checking at compile time instead of run-time
- Can detect type mismatch **BEFORE** your code runs

- Before generics

```
ThingsHolder l;  
l.add(new String("hi"));  
Shape i = (Shape)l.get(0); ————— Crash!
```

- Before generics

```
ThingsHolder l;  
l.add(new String("hi"));  
Shape i = (Shape)l.get(0);
```

- Alternative

```
ThingsHolder<String> l;  
l.add(new String("hi"));  
Shape i = (Shape)l.get(0);  Compile error
```

- Before generics

```
ThingsHolder l;  
l.add(new String("hi"));  
Shape i = (Shape)l.get(0);
```

- Alternative

```
ThingsHolder<String> l;  
l.add(new String("hi"));  
Shape i = (Shape)l.get(0);  Compile error
```

compile-time errors are always better than run-time!

# Primitive Types and Generics

- Generics only work with reference types
  - no int, char, float, double, ...
- What if we need an ArrayList of ints?
- Java has “wrapper” classes
  - Integer, Float, Double
  - These are reference types containing a single primitive...
  - ...and methods to access it
    - intValue(), doubleValue()

- Java will automatically insert the appropriate code to convert between primitive/reference

```
ArrayList<Integer> l;
```

```
l.add(5);
```

```
int i = l.get(n);
```

- Java will automatically insert the appropriate code to convert between primitive/reference

```
ArrayList<Integer> l;  
l.add(5); —————→ l.add(new Integer(5));  
  
int i = l.get(n);  
————→  
int i = l.get(n).intValue();
```

# Generic static methods

- static methods can have their own generic types
- Declare the generic type before the return type:

```
public static <T> boolean doWork(...) { ... }
```

- We can refer to  $T$  as a type within that method only!

# Generic static methods

```
public static <T> boolean contains(T[] array, T item)
{
    for(int i=0; i < array.length;
        i++) if(array[i].equals(item))
            return true;
    return false;
}
```

# Reading

- <https://docs.oracle.com/javase/tutorial/java/generics/types.html>

**Next Time...**

- Recursion and searching
  - Reading: Chapter 12-13.1
- New assignment is up
  - Follow instructions exactly, otherwise, you'll lose points
  - Start your assignment early!

# Quiz time!

# Object Oriented Programming

## CSC220 | Computer Programming 2

# Object-Oriented Programming (OOP)

# Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects  
(or classes) rather than as a set of operations

Why?

# Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects  
(or classes) rather than as a set of operations

Why?

Reusability!

# Definition of Object

- The encapsulation of two things: object data and object behaviors
- Attributes (data) and methods (behaviors)
- Examples: each individual human being, the Costco store at West Kendall Miami, your first car, and the coach bag you are using, etc.

# Definition of Class

- A class is a blueprint for an object
- Class comes first, then object
- Without a class, an object cannot be instantiated

# Human Beings

Class, blueprint for human

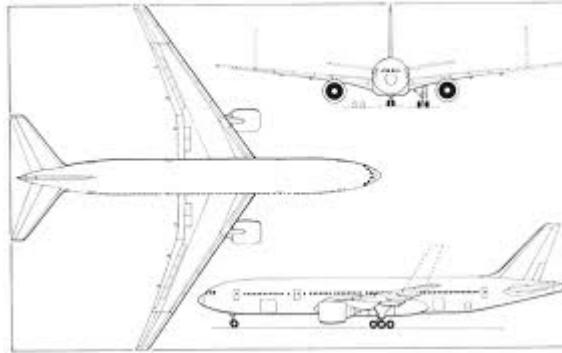


| Human                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------|
| -name: String<br>-gender: String<br>-ssn: String<br>-eyeColor: int                                                                               |
| -walk(): void<br>-sleep(): void<br>-fall_in_love(person)<br>+getName(): String<br>+setName(): String<br>+getSSN(): String<br>+getEyeColor(): int |



# Boeing 787 - Dreamliner

Class, blueprint for 787



Boeing787

-length: float  
-numOfEngine: int  
-numPassengers: int  
-airline: String

-takeOff(): void  
-land: void  
+setAirline(): void  
+getAirline(): String  
-setSerialNum(): void  
+getSerialNum(): String



Object 1,  
The prototype used for flight testing  
during 2009-2010



Object 2,  
The first 787 for Korean  
Airline in February 2012



Object 3,  
The first 787 for United  
Airline at early 2012

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Keyword **public** is an **access modifier**.
  - Indicates that the class is “available to the public”
- Each class declaration that begins with keyword **public** must be stored in a file that has the same name as the class and ends with the **.java** file-name extension.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- By convention, class name begins with a capital letter.

# A Boeing787 Class

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10 }  
11 }
```

- Instance variable: Every instance (i.e., object) of a class contains one copy of each instance variable in the memory
- For a String variable, its initial value is null

# A Boeing787 Class

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10 }  
11 }
```



- Instance variables typically declared as private.
  - **private** is an **access modifier**.
  - private variables and methods are accessible only to methods of the class in which they are declared.
- Declaring instance private is known as **data hiding** or information hiding.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- **private** variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
  - Prevents instance variables from being modified accidentally by a class in another part of the program.
  - *Set* and *get* methods used to access instance variables.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- String is a type.
- other types include: int, char, float, double, and boolean.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10}
11}
```

- A **public** is “available to the public”
  - It can be called from methods of other classes.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- “void” is the return type.
- The **return type** specifies the type of data the method returns after performing its task.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- “void” is the return type.
- The **return type** specifies the type of data the method returns after performing its task.
- Return type **void** indicates that a method will perform a task but will *not* return (i.e., give back) any information to its **calling method** when it completes its task.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Parameter list of the method
- Details later

# A Boeing787 Class

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9 }  
10  
11 }
```

- Notice that the method `displayMessage` accesses the private instance variable “`airLine`”.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Save it as `Boeing787.java` and run it by:

`javac Boeing787.java`

`java Boeing787`

- What would happen?

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Can't execute **Boeing787**; will receive an error message like:
  - Exception in thread "main"  
java.lang.NoSuchMethodError: main
- Class **Boeing787** is not an application because it does not contain **main**.

# A Boeing787 Class

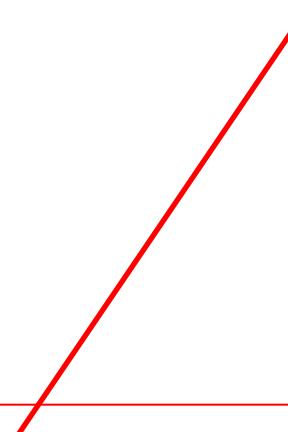
```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10 }  
11 }
```

- Must either declare a separate class that contains a **main** method or place a **main** method in class **Boeing787**.
- The **main** method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- To help you prepare for the larger programs, use a separate class containing method **main** to test each new class.
- Some programmers refer to such a class as a *driver class*.
- Every class can have a **main** method.
- How about if you have multiple **main** method?

# The class with **main** Method

- Define another class FlightSimulation as the application's driver class

```
1
2 public class FlightSimulation {
3
4@    public static void main(String[] args) {
5        //print welcome information
6        System.out.println("Welcome to flight simulation system");
7
8        Boeing787 myB787 = new Boeing787();
9        myB787.displayMessage();
10
11    }
12
13 }
14
```



# The class with **main** Method

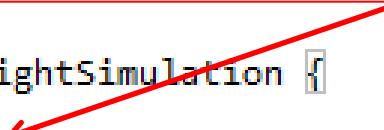
- Can you call a method of a class without creating an object of that class?

```
1 public class FlightSimulation {  
2  
3  
4@    public static void main(String[] args) {  
5        //print welcome information  
6        System.out.println("Welcome to flight simulation system");  
7  
8        Boeing787 myB787 = new Boeing787();  
9        myB787.displayMessage();  
10  
11    }  
12  
13 }  
14
```

# The class with **main** Method

- A **static** method (such as **main**) is special
  - It can be called without first creating an object of the class in which the method is declared.
- Typically, you cannot call a method that belongs to another class until you create an object of that class.

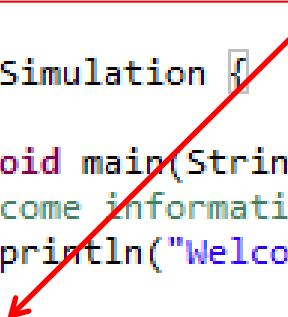
```
1
2 public class FlightSimulation {
3
4@    public static void main(String[] args) {
5        //print welcome information
6        System.out.println("Welcome to flight simulation system");
7
8        Boeing787 myB787 = new Boeing787();
9        myB787.displayMessage();
10
11    }
12
13 }
14
```



# The class with **main** Method

- Declare a variable of the class type.
  - Each new class you create becomes a new type that can be used to declare variables and create objects.
  - You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.

```
1  public class FlightSimulation {  
2  
3  
4@  public static void main(String[] args) {  
5      //print welcome information  
6      System.out.println("Welcome to flight simulation system");  
7  
8      Boeing787 myB787 = new Boeing787();  
9      myB787.displayMessage();  
10  
11  }  
12  
13 }  
14
```



# The class with **main** Method

- Invoke or call a method of an object
  - Variable name followed by a **dot separator** (.), the method name and parentheses.
  - Call causes the method to perform its task.

```
1
2 public class FlightSimulation {
3
4@  public static void main(String[] args) {
5      //print welcome information
6      System.out.println("Welcome to flight simulation system");
7
8      Boeing787 myB787 = new Boeing787();
9      myB787.displayMessage();
10
11  }
12
13 }
14
```

# What would be the output?

```
1 public class FlightSimulation {  
2     //  
3     public static void main(String[] args) {  
4         //print welcome information  
5         System.out.println("Welcome to flight simulation system");  
6     }  
7     //  
8     Boeing787 myB787 = new Boeing787();  
9     myB787.displayMessage();  
10    }  
11 }  
12 //  
13 //  
14 }
```

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4     //  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9 }  
10 }  
11 }
```

# *get* and *set* Methods

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11    //set method
12    public void setAirLine(String nameArg){
13        airLine = nameArg;
14    }
15
16    //get method
17    public String getAirLine(){
18        return airLine;
19    }
20
21 }
```

## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- A method can require one or more parameters that represent additional information it needs to perform its task.
  - Defined in a comma-separated **parameter list**
  - Located in the parentheses that follow the method name
  - Each parameter must specify a type and an identifier.

## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- Local variables

- Variables declared in the body of a particular method.
- When a method terminates, the values of its local variables are lost.

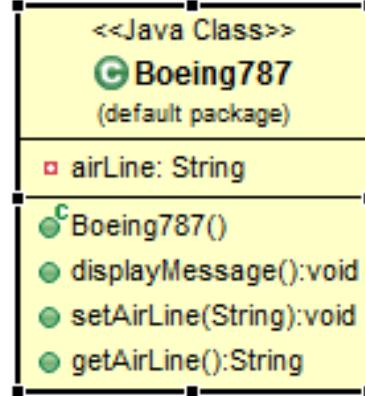
## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- Return a String type value

# Unified Modeling Language (UML) Class Diagram

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10    //set method  
11    public void setAirLine(String nameArg){  
12        airLine = nameArg;  
13    }  
14  
15    //get method  
16    public String getAirLine(){  
17        return airLine;  
18    }  
19  
20}  
21 }
```



The diagram shows a UML class named 'Boeing787' in the 'default package'. It has one attribute, 'airLine', which is a String type. It also has four operations: a constructor 'Boeing787()', and three methods: 'displayMessage():void', 'setAirLine(String):void', and 'getAirLine():String'.

```

1 import java.util.Scanner; ←
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airline = input.nextLine(); ←
15        myB787.setAirLine(airline);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }

```

- Import class Scanner

- Why not import System and String class in package java.lang?
- Java.lang is implicitly imported into every Java program

```

1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }

```

```

1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }

```

- Why not import while using Boeing 787?
- It is under the same folder as FlightSimulation.java
- Default package
- No need to import

```

1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }

```

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }
}
```

Must Match:  
Type  
number

Argument

Parameter

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }
```

# Can one file contain multiple classes?

- Yes.
- Nested class

```
class OuterClass{  
    ...  
    class NestedClass{  
        ...  
    }  
}
```

- More information:  
<http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html>

# Can one file contain multiple classes?

- You **cannot** do this:

```
public class A{  
    ...  
}
```

```
public class B{  
    ...  
}
```

```
|  
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
|   The public type CounterTerrorist must be defined in its own file  
|  
|       at CounterTerrorist.<init>(Terrorist.java:54)  
|       at CS.main(CS.java:23)
```

# This is what you should do: One file contains one class

- Readability
- Easy to control and collaborate
- Java has strict rules
- If Java sees enabling something can bring potentials of more troubles than convenience, it will not allow it.
- E.g. no multiple inheritance (C++ allowed)

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }
}
```

FlightSimulation.java

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }
```

Boeing787.java

# Primitive Types vs. Reference Types

- Primitive types includes `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- Primitive type variable that initialized as 0: `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- Primitive type variable that initialized as false: `boolean`



**Fixed amount of memory,  
Java handles it directly, or by value.**

# Primitive Types vs. Reference Types

- All nonprimitive types are reference types.
- Classes, which specify the types of objects, are reference types.
- Programs use variables of reference types (normally called **references**) to store the locations of objects in the computer's memory.
  - Such a variable is said to refer to an object in the program.
- Objects that are referenced may each contain many instance variables and methods.

```
Boeing787 myB787 = new Boeing787();
```

No standard size in memory,  
Java handles them indirectly, or by reference

## Reference Types

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19    }
20 }
```

Reference-type variable  
(also called  
“Reference”)

Store the locations of  
objects in the  
computer’s memory.

*myB787* is a reference to  
that *Boeing787* object.

```
1 public class Boeing787 {
2     //The attribute of the class
3     private String airLine;
4
5     //The method of the class
6     public void displayMessage(){
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
8     }
9
10    //set method
11    public void setAirLine(String nameArg){
12        airLine = nameArg;
13    }
14
15    //get method
16    public String getAirLine(){
17        return airLine;
18    }
19
20 }
```

When using an object of  
another class, a reference  
to the object is required to  
**invoke** (i.e., call) its  
methods.

Also known as sending  
messages (including  
argument(s)) to an object.

# Difference between Java and C or C++

- Java does NOT have pointer (no incremented or decremented, no converting from value to pointer)
- In C++, calling a method of an object from pointer can be done by using `->`; java: from reference, can be done by using `dot (.)`.
- In C, `malloc ()` and `free ()`
- In C++, `delete` can be used to delete object
- In Java, garbage collection

```

1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }

```

```

1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }

```

## Output:

```

Welcome to flight simulation system
Hello, I am a Boeing 787 belongs to null

Please tell me the airline.
Delta
The airline is Delta

```

## Constructor

- Used to initialize the instance variables while creating an object
- Must be the same as the class name
- Java compiler provides a ***default constructor*** with no parameter in any class that does not include a constructor
- When a class has only default constructor, instance variables were set with default values
- Multiple constructors may be defined, taking different parameters

## When ***new***

- Requests memory
- Calls constructor
- Reclaim garbage collection

```

1 //File name: FlightSimulation.java
2 public class FlightSimulation {
3
4     public static void main(String[] args) {
5         String airLine = "United";
6         double topSpeed = 587;
7         //creating an object with values initialized
8         Boeing787 myB787 = new Boeing787(airLine, topSpeed);
9         myB787.displayMessage();
10        //creating another object
11        Boeing787 myB787_2 = new Boeing787("AA", 587);
12        myB787_2.displayMessage();
13    }
14 }

```

```

1 //File Name: Boeing787.java
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5     private double maxSpeed;
6
7     //Constructor
8     public Boeing787(String airLinePara, double maxSpeedPara){
9         setMaxSpeed(maxSpeedPara);
10        setAirLine(airLinePara);
11    }
12    //set method for the maxSpeed
13    public void setMaxSpeed(double maxSpeedPara){
14        maxSpeed = maxSpeedPara;
15    }
16    //The displaying method of the class
17    public void displayMessage(){
18        System.out.printf("Hello, I am a Boeing 787 belongs to %s\n", airLine);
19        System.out.println("My top speed is " + maxSpeed);
20    }
21    //set method for airLine
22    public void setAirLine(String nameArg){
23        airLine = nameArg;
24    }
25    //get method for airLine
26    public String getAirLine(){
27        return airLine;
28    }
29 }

```

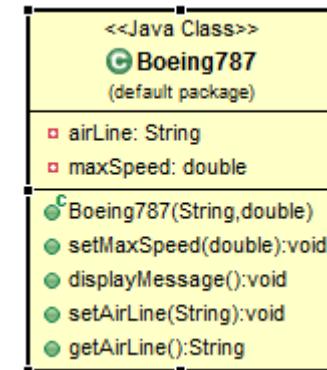
## Output:

```

Hello, I am a Boeing 787 belongs to United
My top speed is 587.0
Hello, I am a Boeing 787 belongs to AA
My top speed is 587.0

```

- Constructors cannot return values, so they cannot specify a return type. NO **void**.
- If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.
- Normally, constructors are declared **public**.



# Garbage Collection of Java

- When you **new** an object, java automatically determines the size of memory the object needs.
- Meanwhile, Java claims that amount of memory for reuse when no reference exists for the object.

```
Boeing787 myB787 = new Boeing787();  
myB787.displayMessage();  
myB787 = new Boeing787();
```

# String Concatenation

- “hello” + “world”, you get “hello world”
- Primitive values are converted to strings if included in string concatenation
- All objects have a *toString()* method that returns a String representation of the object

```
2 public class StringConcatenation {  
3  
4    public static void main(String[] args) {  
5        double d1 = 0.33325;  
6        boolean b1 = true;  
7  
8        System.out.println("In string formate " + d1 + " " + b1);  
9    }  
10   }  
11 }  
12 }
```

```
public class Point{  
    int x;  
    int y;  
  
    public void translate(int dx, int dy){  
        ...  
    }  
}
```

---

```
public static void main(String[] args){  
  
    // create a Point object  
    Point p1 = new Point();  
  
    p1.translate(6, 3); —————→ Call the instance method  
    System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
}
```

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```



i is 36

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```

The diagram shows three arrows pointing from the variable names in the code to their respective values. The first arrow points from 'i' to '36'. The second arrow points from 's' to 'CSC220'. The third arrow points from 'p1' to 'Point()'.

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```

i is 36  
s is CSC220  
p is: PointMain.Point@677327b6

# The `toString` method

- An instance method that returns a `String` representation of the object.

# The `toString` method

```
public String toString() {  
    <code to produce and return the desired String>  
}
```

# The `toString` method

```
public String toString() {  
    <code to produce and return the desired String>  
}
```

```
public String toString(){  
    return x + " " + y;  
}
```

# The `toString` method

```
public class Point{  
    int x;  
    int y;  
  
    public String toString(){  
        return x + " " + y;  
    }  
  
}
```

---

```
public static void main(String[] args){  
  
    int i = 36;  
    String s = "CSC220";  
    Point p1 = new Point();  
  
    System.out.println("i is: " + i);  
    System.out.println("s is: " + s);  
    System.out.println("p is: " + p1); —————→ p is: 00  
}
```

Next Time...

- **First quiz** on Thursday

- Don't be late!
- Mainly from OOP
- You have a lab tomorrow
- New lab assignment and homework assignment
- Continue the introduction to OOP
  - More about object and reference semantic
  - If time permits...
    - Learn about ArrayList (Chapter 10.1)

# Object Oriented Programming

## CSC220 | Computer Programming 2

# Object-Oriented Programming (OOP)

# Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects  
(or classes) rather than as a set of operations

Why?

# Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects  
(or classes) rather than as a set of operations

Why?

Reusability!

# Definition of Object

- The encapsulation of two things: object data and object behaviors
- Attributes (data) and methods (behaviors)
- Examples: each individual human being, the Costco store at West Kendall Miami, your first car, and the coach bag you are using, etc.

# Definition of Class

- A class is a blueprint for an object
- Class comes first, then object
- Without a class, an object cannot be instantiated

# Human Beings

Class, blueprint for human

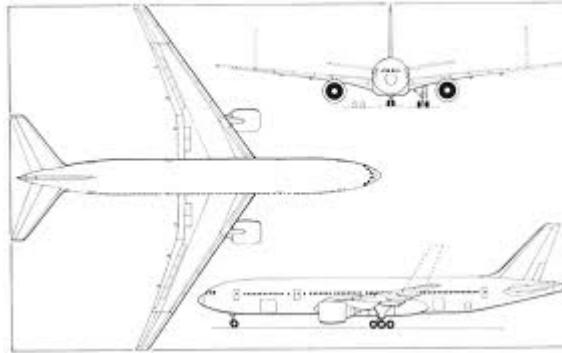


| Human                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------|
| -name: String<br>-gender: String<br>-ssn: String<br>-eyeColor: int                                                                               |
| -walk(): void<br>-sleep(): void<br>-fall_in_love(person)<br>+getName(): String<br>+setName(): String<br>+getSSN(): String<br>+getEyeColor(): int |



# Boeing 787 - Dreamliner

Class, blueprint for 787



Boeing787

-length: float  
-numOfEngine: int  
-numPassengers: int  
-airline: String

-takeOff(): void  
-land: void  
+setAirline(): void  
+getAirline(): String  
-setSerialNum(): void  
+getSerialNum(): String



Object 1,  
The prototype used for flight testing  
during 2009-2010



Object 2,  
The first 787 for Korean  
Airline in February 2012



Object 3,  
The first 787 for United  
Airline at early 2012

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Keyword **public** is an **access modifier**.
  - Indicates that the class is “available to the public”
- Each class declaration that begins with keyword **public** must be stored in a file that has the same name as the class and ends with the **.java** file-name extension.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- By convention, class name begins with a capital letter.

# A Boeing787 Class

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10 }  
11 }
```

- Instance variable: Every instance (i.e., object) of a class contains one copy of each instance variable in the memory
- For a String variable, its initial value is null

# A Boeing787 Class

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9 }  
10  
11 }
```



- Instance variables typically declared as private.
  - **private** is an **access modifier**.
  - private variables and methods are accessible only to methods of the class in which they are declared.
- Declaring instance private is known as **data hiding** or information hiding.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- **private** variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
  - Prevents instance variables from being modified accidentally by a class in another part of the program.
  - *Set* and *get* methods used to access instance variables.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- String is a type.
- other types include: int, char, float, double, and boolean.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- A **public** is “available to the public”
  - It can be called from methods of other classes.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- “void” is the return type.
- The **return type** specifies the type of data the method returns after performing its task.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- “void” is the return type.
- The **return type** specifies the type of data the method returns after performing its task.
- Return type **void** indicates that a method will perform a task but will *not* return (i.e., give back) any information to its **calling method** when it completes its task.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Parameter list of the method
- Details later

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Notice that the method `displayMessage` accesses the private instance variable “`airLine`”.

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Save it as `Boeing787.java` and run it by:

`javac Boeing787.java`

`java Boeing787`

- What would happen?

# A Boeing787 Class

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11 }
```

- Can't execute **Boeing787**; will receive an error message like:
  - Exception in thread "main"  
java.lang.NoSuchMethodError: main
- Class **Boeing787** is not an application because it does not contain **main**.

# A Boeing787 Class

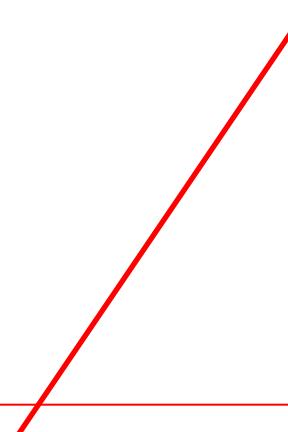
```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10 }  
11 }
```

- Must either declare a separate class that contains a **main** method or place a **main** method in class **Boeing787**.
- The **main** method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- To help you prepare for the larger programs, use a separate class containing method **main** to test each new class.
- Some programmers refer to such a class as a *driver class*.
- Every class can have a **main** method.
- How about if you have multiple **main** method?

# The class with **main** Method

- Define another class FlightSimulation as the application's driver class

```
1
2 public class FlightSimulation {
3
4@    public static void main(String[] args) {
5        //print welcome information
6        System.out.println("Welcome to flight simulation system");
7
8        Boeing787 myB787 = new Boeing787();
9        myB787.displayMessage();
10
11    }
12
13 }
14
```



# The class with **main** Method

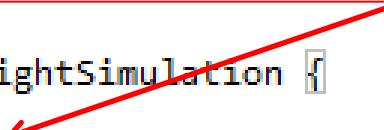
- Can you call a method of a class without creating an object of that class?

```
1 public class FlightSimulation {  
2  
3  
4@    public static void main(String[] args) {  
5        //print welcome information  
6        System.out.println("Welcome to flight simulation system");  
7  
8        Boeing787 myB787 = new Boeing787();  
9        myB787.displayMessage();  
10  
11    }  
12  
13 }  
14
```

# The class with **main** Method

- A **static** method (such as **main**) is special
  - It can be called without first creating an object of the class in which the method is declared.
- Typically, you cannot call a method that belongs to another class until you create an object of that class.

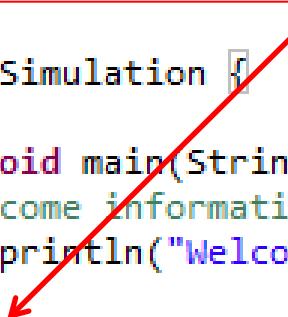
```
1
2 public class FlightSimulation {
3
4@    public static void main(String[] args) {
5        //print welcome information
6        System.out.println("Welcome to flight simulation system");
7
8        Boeing787 myB787 = new Boeing787();
9        myB787.displayMessage();
10
11    }
12
13 }
14
```



# The class with **main** Method

- Declare a variable of the class type.
  - Each new class you create becomes a new type that can be used to declare variables and create objects.
  - You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.

```
1  public class FlightSimulation {  
2  
3  
4@  public static void main(String[] args) {  
5      //print welcome information  
6      System.out.println("Welcome to flight simulation system");  
7  
8      Boeing787 myB787 = new Boeing787();  
9      myB787.displayMessage();  
10  
11  }  
12  
13 }  
14
```



# The class with **main** Method

- Invoke or call a method of an object
  - Variable name followed by a **dot separator** (.), the method name and parentheses.
  - Call causes the method to perform its task.

```
1
2 public class FlightSimulation {
3
4@  public static void main(String[] args) {
5      //print welcome information
6      System.out.println("Welcome to flight simulation system");
7
8      Boeing787 myB787 = new Boeing787();
9      myB787.displayMessage();
10
11  }
12
13 }
14
```

# What would be the output?

```
1 public class FlightSimulation {  
2     //  
3     public static void main(String[] args) {  
4         //print welcome information  
5         System.out.println("Welcome to flight simulation system");  
6     }  
7     //  
8     Boeing787 myB787 = new Boeing787();  
9     myB787.displayMessage();  
10    }  
11 }  
12 //  
13 //  
14 }
```

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4     //  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9 }  
10 }  
11 }
```

# *get* and *set* Methods

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11    //set method
12    public void setAirLine(String nameArg){
13        airLine = nameArg;
14    }
15
16    //get method
17    public String getAirLine(){
18        return airLine;
19    }
20
21 }
```

## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- A method can require one or more parameters that represent additional information it needs to perform its task.
  - Defined in a comma-separated **parameter list**
  - Located in the parentheses that follow the method name
  - Each parameter must specify a type and an identifier.

## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- Local variables

- Variables declared in the body of a particular method.
- When a method terminates, the values of its local variables are lost.

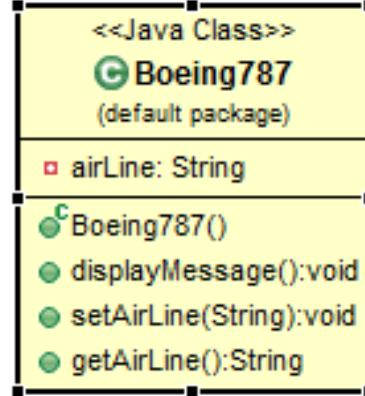
## *get* and *set* Methods

```
//set method  
public void setAirLine(String nameArg){  
    airLine = nameArg;  
}  
  
//get method  
public String getAirLine(){  
    return airLine;  
}
```

- Return a String type value

# Unified Modeling Language (UML) Class Diagram

```
1 public class Boeing787 {  
2     //The attribute of the class  
3     private String airLine;  
4  
5     //The method of the class  
6     public void displayMessage(){  
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);  
8     }  
9  
10    //set method  
11    public void setAirLine(String nameArg){  
12        airLine = nameArg;  
13    }  
14  
15    //get method  
16    public String getAirLine(){  
17        return airLine;  
18    }  
19  
20}  
21 }
```



The diagram shows a UML class named 'Boeing787' in the default package. It has one attribute, 'airLine', which is a String type. It features four operations: a constructor 'Boeing787()', a display message operation 'displayMessage():void', a set attribute operation 'setAirLine(String):void', and a get attribute operation 'getAirLine():String'.

```

1 import java.util.Scanner; ←
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airline = input.nextLine(); ←
15        myB787.setAirLine(airline);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }

```

- Import class Scanner

- Why not import System and String class in package java.lang?
- Java.lang is implicitly imported into every Java program

```

1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }

```

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }
}
```

- Why not import while using Boeing 787?
- It is under the same folder as FlightSimulation.java
- Default package
- No need to import

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }
```

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }
}
```

Must Match:  
Type  
number

Argument

Parameter

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }
```

# Can one file contain multiple classes?

- Yes.
- Nested class

```
class OuterClass{  
    ...  
    class NestedClass{  
        ...  
    }  
}
```

- More information:  
<http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html>

# Can one file contain multiple classes?

- You **cannot** do this:

```
public class A{  
    ...  
}
```

```
public class B{  
    ...  
}
```

```
|  
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
|   The public type CounterTerrorist must be defined in its own file  
|  
|       at CounterTerrorist.<init>(Terrorist.java:54)  
|       at CS.main(CS.java:23)
```

# This is what you should do: One file contains one class

- Readability
- Easy to control and collaborate
- Java has strict rules
- If Java sees enabling something can bring potentials of more troubles than convenience, it will not allow it.
- E.g. no multiple inheritance (C++ allowed)

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }
}
```

FlightSimulation.java

```
1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }
```

Boeing787.java

# Primitive Types vs. Reference Types

- Primitive types includes `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- Primitive type variable that initialized as 0: `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- Primitive type variable that initialized as false: `boolean`



**Fixed amount of memory,  
Java handles it directly, or by value.**

# Primitive Types vs. Reference Types

- All nonprimitive types are reference types.
- Classes, which specify the types of objects, are reference types.
- Programs use variables of reference types (normally called **references**) to store the locations of objects in the computer's memory.
  - Such a variable is said to refer to an object in the program.
- Objects that are referenced may each contain many instance variables and methods.

```
Boeing787 myB787 = new Boeing787();
```

No standard size in memory,  
Java handles them indirectly, or by reference

## Reference Types

```
1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19    }
20 }
```

Reference-type variable  
(also called  
“Reference”)

Store the locations of  
objects in the  
computer’s memory.

*myB787* is a reference to  
that *Boeing787* object.

```
1 public class Boeing787 {
2     //The attribute of the class
3     private String airLine;
4
5     //The method of the class
6     public void displayMessage(){
7         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
8     }
9
10    //set method
11    public void setAirLine(String nameArg){
12        airLine = nameArg;
13    }
14
15    //get method
16    public String getAirLine(){
17        return airLine;
18    }
19
20 }
```

When using an object of  
another class, a reference  
to the object is required to  
**invoke** (i.e., call) its  
methods.

Also known as sending  
messages (including  
argument(s)) to an object.

# Difference between Java and C or C++

- Java does NOT have pointer (no incremented or decremented, no converting from value to pointer)
- In C++, calling a method of an object from pointer can be done by using `->`; java: from reference, can be done by using `dot (.)`.
- In C, `malloc ()` and `free ()`
- In C++, `delete` can be used to delete object
- In Java, garbage collection

```

1 import java.util.Scanner;
2
3 public class FlightSimulation {
4
5     public static void main(String[] args) {
6         //print welcome information
7         System.out.println("Welcome to flight simulation system");
8
9         Boeing787 myB787 = new Boeing787();
10        myB787.displayMessage();
11
12        System.out.print("Please tell me the airline.\n");
13        Scanner input = new Scanner(System.in);
14        String airLine = input.nextLine();
15        myB787.setAirLine(airLine);
16        System.out.println("The airline is " + myB787.getAirLine() + "\n");
17
18        input.close();
19
20    }

```

```

1
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5
6     //The method of the class
7     public void displayMessage(){
8         System.out.printf("Hello, I am a Boeing 787 belongs to %s\n\n", airLine);
9     }
10
11     //set method
12     public void setAirLine(String nameArg){
13         airLine = nameArg;
14     }
15
16     //get method
17     public String getAirLine(){
18         return airLine;
19     }
20
21 }

```

## Output:

```

Welcome to flight simulation system
Hello, I am a Boeing 787 belongs to null

Please tell me the airline.
Delta
The airline is Delta

```

## Constructor

- Used to initialize the instance variables while creating an object
- Must be the same as the class name
- Java compiler provides a ***default constructor*** with no parameter in any class that does not include a constructor
- When a class has only default constructor, instance variables were set with default values
- Multiple constructors may be defined, taking different parameters

## When ***new***

- Requests memory
- Calls constructor
- Reclaim garbage collection

```

1 //File name: FlightSimulation.java
2 public class FlightSimulation {
3
4     public static void main(String[] args) {
5         String airLine = "United";
6         double topSpeed = 587;
7         //creating an object with values initialized
8         Boeing787 myB787 = new Boeing787(airLine, topSpeed);
9         myB787.displayMessage();
10        //creating another object
11        Boeing787 myB787_2 = new Boeing787("AA", 587);
12        myB787_2.displayMessage();
13    }
14 }

```

```

1 //File Name: Boeing787.java
2 public class Boeing787 {
3     //The attribute of the class
4     private String airLine;
5     private double maxSpeed;
6
7     //Constructor
8     public Boeing787(String airLinePara, double maxSpeedPara){
9         setMaxSpeed(maxSpeedPara);
10        setAirLine(airLinePara);
11    }
12    //set method for the maxSpeed
13    public void setMaxSpeed(double maxSpeedPara){
14        maxSpeed = maxSpeedPara;
15    }
16    //The displaying method of the class
17    public void displayMessage(){
18        System.out.printf("Hello, I am a Boeing 787 belongs to %s\n", airLine);
19        System.out.println("My top speed is " + maxSpeed);
20    }
21    //set method for airLine
22    public void setAirLine(String nameArg){
23        airLine = nameArg;
24    }
25    //get method for airLine
26    public String getAirLine(){
27        return airLine;
28    }
29 }

```

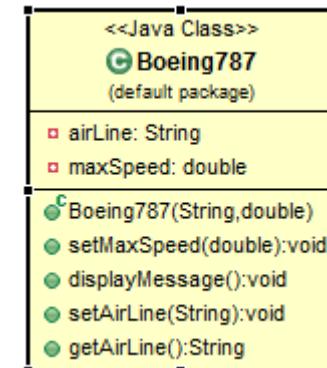
## Output:

```

Hello, I am a Boeing 787 belongs to United
My top speed is 587.0
Hello, I am a Boeing 787 belongs to AA
My top speed is 587.0

```

- Constructors cannot return values, so they cannot specify a return type. NO **void**.
- If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.
- Normally, constructors are declared **public**.



# Garbage Collection of Java

- When you **new** an object, java automatically determines the size of memory the object needs.
- Meanwhile, Java claims that amount of memory for reuse when no reference exists for the object.

```
Boeing787 myB787 = new Boeing787();  
myB787.displayMessage();  
myB787 = new Boeing787();
```

# String Concatenation

- “hello” + “world”, you get “hello world”
- Primitive values are converted to strings if included in string concatenation
- All objects have a *toString()* method that returns a String representation of the object

```
2 public class StringConcatenation {  
3  
4    public static void main(String[] args) {  
5        double d1 = 0.33325;  
6        boolean b1 = true;  
7  
8        System.out.println("In string formate " + d1 + " " + b1);  
9    }  
10   }  
11 }  
12 }
```

```
public class Point{  
    int x;  
    int y;  
  
    public void translate(int dx, int dy){  
        ...  
    }  
}
```

---

```
public static void main(String[] args){  
  
    // create a Point object  
    Point p1 = new Point();  
  
    p1.translate(6, 3); —————→ Call the instance method  
    System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
}
```

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```



i is 36

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```

The code demonstrates the use of the `toString` method. The first three statements print the variables `i`, `s`, and `p1` respectively. The fourth statement prints the `Point` object `p1`. Arrows point from the variable names to their corresponding values: `i` points to `36`, `s` points to `CSC220`, and `p1` points to `Point()`.

# The `toString` method

- An instance method that returns a `String` representation of the object.

```
int i = 36;
String s = "CSC220";
Point p1 = new Point();

System.out.println("i is: " + i);
System.out.println("s is: " + s);
System.out.println("p is: " + p1);
```

i is 36  
s is CSC220  
p is: PointMain.Point@677327b6

# The `toString` method

- An instance method that returns a `String` representation of the object.

# The `toString` method

```
public String toString() {  
    <code to produce and return the desired String>  
}
```

# The `toString` method

```
public String toString() {  
    <code to produce and return the desired String>  
}
```

```
public String toString(){  
    return x + " " + y;  
}
```

# The `toString` method

```
public class Point{  
    int x;  
    int y;  
  
    public String toString(){  
        return x + " " + y;  
    }  
  
}
```

---

```
public static void main(String[] args){  
  
    int i = 36;  
    String s = "CSC220";  
    Point p1 = new Point();  
  
    System.out.println("i is: " + i);  
    System.out.println("s is: " + s);  
    System.out.println("p is: " + p1); —————→ p is: 00  
}
```

Next Time...

- **First quiz** on Thursday

- Don't be late!
- Mainly from OOP
- You have a lab tomorrow
- New lab assignment and homework assignment
- Continue the introduction to OOP
  - More about object and reference semantic
  - If time permits...
    - Learn about ArrayList (Chapter 10.1)

# Object Oriented Programming

## CSC220|Computer Programming 2

this

- this is a reference to the current object

```
public class Point{  
    int x;  
    int y;  
  
    public void translate(int dx, int dy){  
        this.x = dx;  
        this.y += dy;  
    }  
}
```

this

```
public class Point{
    int x;
    int y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

---

```
public static void main(String[] args){

    Point p1 = new Point(3, 5);

    System.out.println(p1.x + " " + p1.y);
}
```

---

?

this

```
public class Point{
    int x;
    int y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

---

```
public static void main(String[] args){

    Point p1 = new Point(3, 5);

    System.out.println(p1.x + " " + p1.y);
}
```

---

3 5

# Reference declaration

- Declaration of a reference variable only provides a name to reference an object
  - It does not create an object
- After `Point p1;` the value stored in `p1` is \_\_\_\_\_

# Reference declaration

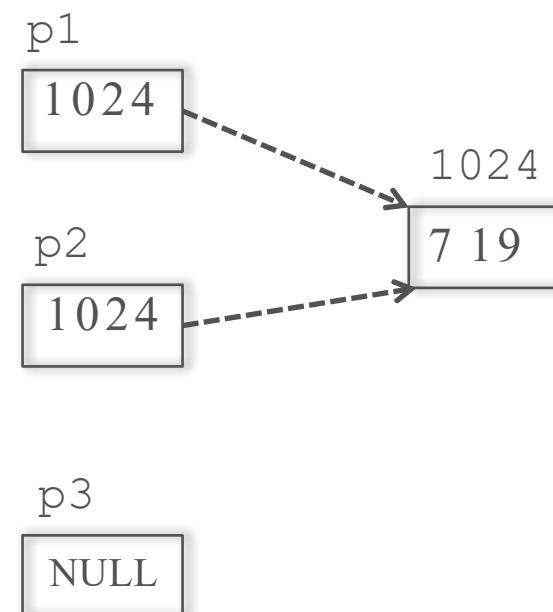
- Declaration of a reference variable only provides a name to reference an object
  - It does not create an object
- After `Point p1;` the value stored in `p1` is null

- All non-primitive types are **reference types**
- A **reference** is a variable that stores the memory address where an object (a group of values) resides

```
Point p1, p2, p3;  
p1 = new Point(7,19);  
p2 = p1;
```

- All non-primitive types are **reference types**
- A **reference** is a variable that stores the memory address where an object (a group of values) resides

```
Point p1, p2, p3;  
p1 = new Point(7,19);  
p2 = p1;
```



# Operations on reference types

- Operations on references: `=`, `==`, `!=`
  - Equality operators compare addresses

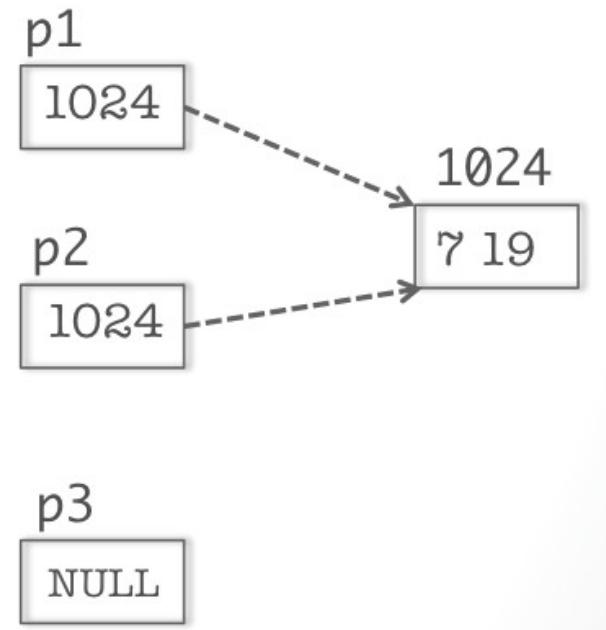
# Operations on reference types

- Operations on references: `=`, `==`, `!=`
  - Equality operators compare addresses
  - What does `p2 == p1` return?

# Operations on reference types

- Operations on references: `=`, `==`, `!=`
  - Equality operators compare addresses
  - What does `p2 == p1` return?

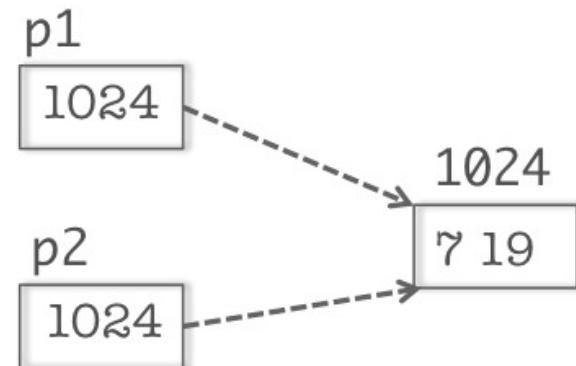
```
Point p1, p2, p3;  
p1 = new Point(7,19);  
p2 = p1;
```



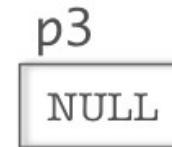
# Operations on reference types

- Operations on references: `=`, `==`, `!=`
  - Equality operators compare addresses
  - What does `p2 == p1` return?

```
Point p1, p2, p3;  
p1 = new Point(7,19);  
p2 = p1;
```



What does `p2.firstValue()` return?  
if `firstValue()` has been implemented  
to return the first value



# Useful reference types

- Strings
- Classes
- Arrays
  - Add something to an array not supported
  - Removing from an array not supported

# ArrayList

# ArrayList

- The `ArrayList` class (from the Collections library) mimics an array and allows for dynamic expansion
- The `get`, `set` methods are used in place of `[]` for indexing

# ArrayList

- the `add` method increases the size by one and adds a new item
- the `remove` method preserves the order of the list by shifting values to the left to fill in any gap.
- `ArrayList` may only be used with reference types

`ArrayList<E>`

# ArrayList

ArrayList<E>

ArrayList<String> list = new ArrayList<String>();

type

type

Point p = new Point();

type

type

# One Last Example

```
public class Account {  
    int dollars;  
  
    public Account(int Initialdollars){  
        dollars = Initialdollars;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer1( Account rhs)  
    {  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer2( Account rhs)  
    {  
        if (this == rhs)  
            return;  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
}
```

---

```
public class Account {  
    int dollars;  
  
    public Account(int Initialdollars){  
        dollars = Initialdollars;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer1( Account rhs)  
    {  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer2( Account rhs)  
    {  
        if (this == rhs)  
            return;  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
}  
  


---

  
public static void main(String[] args){  
  
    Account account1 = new Account(100);  
    Account account2 = new Account(200);  
  
    account1.finalTransfer1( account2);  
  
    System.out.println("account1: " + account1.dollars);  
    System.out.println("account2: " + account2.dollars);  
}
```

```

public class Account {
    int dollars;

    public Account(int Initialdollars){
        dollars = Initialdollars;
    }

    // transfer all money from rhs to current account
    public void finalTransfer1( Account rhs)
    {
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }

    // transfer all money from rhs to current account
    public void finalTransfer2( Account rhs)
    {
        if (this == rhs)
            return;
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }
}

```

---

```

public static void main(String[] args){
    Account account1 = new Account(100);
    Account account2 = new Account(200);

    account1.finalTransfer1( account2);

    System.out.println("account1: " + account1.dollars);
    System.out.println("account2: " + account2.dollars);
}

```



```
public class Account {  
    int dollars;  
  
    public Account(int Initialdollars){  
        dollars = Initialdollars;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer1( Account rhs)  
    {  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
  
    // transfer all money from rhs to current account  
    public void finalTransfer2( Account rhs)  
    {  
        if (this == rhs)  
            return;  
        dollars += rhs.dollars;  
        rhs.dollars = 0;  
    }  
}  
  


---

  
public static void main(String[] args){  
  
    Account account1 = new Account(100);  
    Account account2 = new Account(200);  
  
    account2 = account1;  
    account1.finalTransfer1( account2);  
  
    System.out.println("account1: " + account1.dollars);  
    System.out.println("account2: " + account2.dollars);  
}
```

```

public class Account {
    int dollars;

    public Account(int Initialdollars){
        dollars = Initialdollars;
    }

    // transfer all money from rhs to current account
    public void finalTransfer1( Account rhs)
    {
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }

    // transfer all money from rhs to current account
    public void finalTransfer2( Account rhs)
    {
        if (this == rhs)
            return;
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }
}

```

```

public static void main(String[] args){
    Account account1 = new Account(100);
    Account account2 = new Account(200);

    account2 = account1;
    account1.finalTransfer1( account2);

    System.out.println("account1: " + account1.dollars);
    System.out.println("account2: " + account2.dollars);
}

```

Account1: 0

Account2: 0

Account1

1024

Account2

1024

1024

100

1036

200

```

public class Account {
    int dollars;

    public Account(int Initialdollars){
        dollars = Initialdollars;
    }

    // transfer all money from rhs to current account
    public void finalTransfer1( Account rhs)
    {
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }

    // transfer all money from rhs to current account
    public void finalTransfer2( Account rhs)
    {
        if (this == rhs)
            return;
        dollars += rhs.dollars;
        rhs.dollars = 0;
    }
}

```

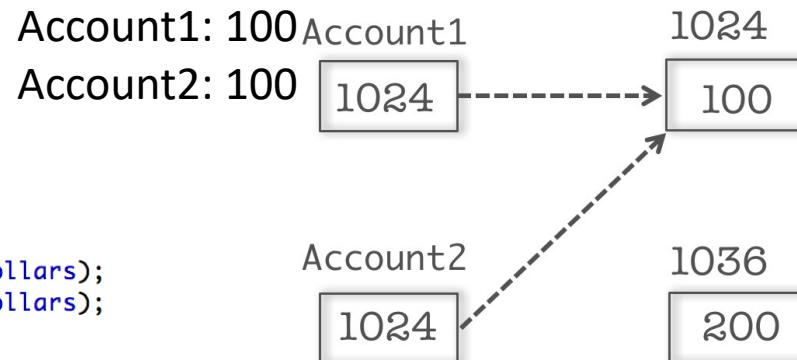
```

public static void main(String[] args){
    Account account1 = new Account(100);
    Account account2 = new Account(200);

    account2 = account1;
    account1.finalTransfer2( account2);

    System.out.println("account1: " + account1.dollars);
    System.out.println("account2: " + account2.dollars);
}

```

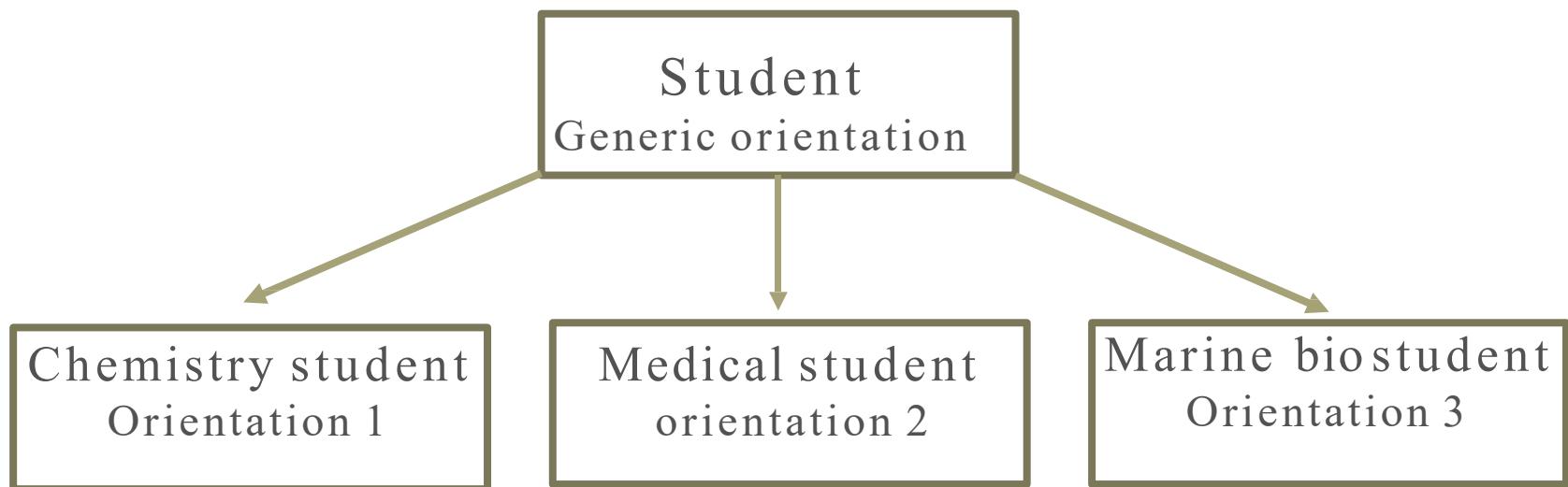


# Inheritance and Interfaces

# Inheritance and interfaces

- A feature of OOP
- Reusability
  - Allow one class to be an extension of another
- Code on scale
  - Write programs with hierarchies of related object types.

# A nonprogramming example



There is natural hierarchy

Or...

# A nonprogramming example

Chemistry student  
Orientation 1  
(covers generic rules)

Medical student  
orientation 2  
(covers generic rules)

Marine bio student  
Orientation 3  
(covers generic rules)

# A nonprogramming example

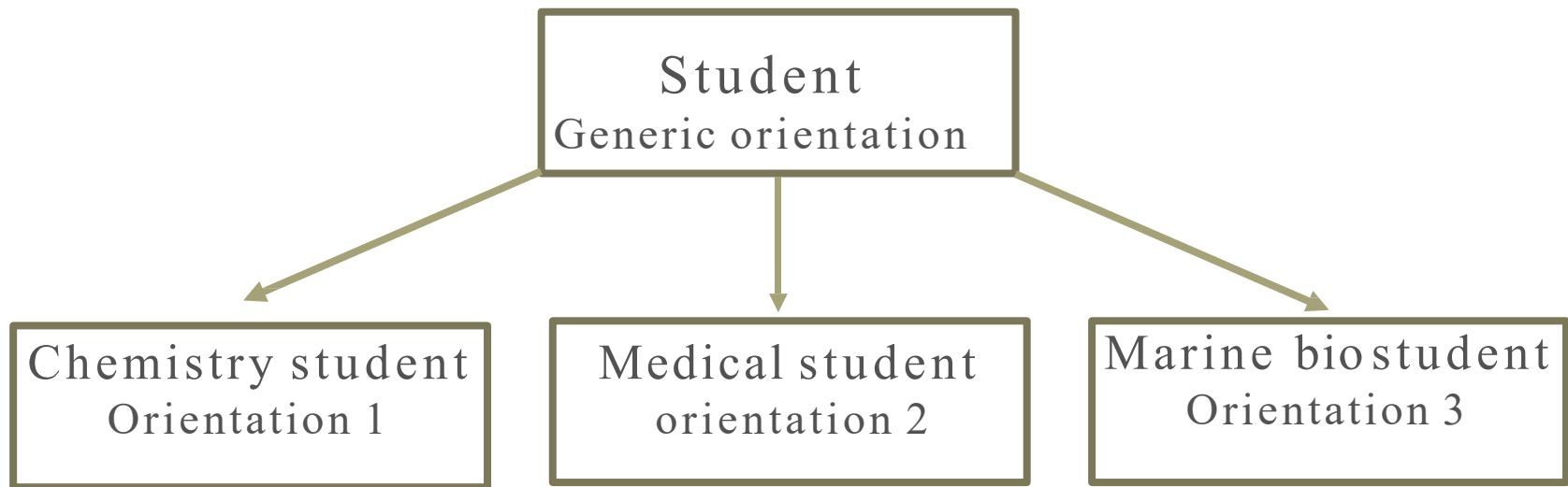
Chemistry student  
Orientation 1  
(covers generic rules)

Medical student  
orientation 2  
(covers generic rules)

Marine bio student  
Orientation 3  
(covers generic rules)

What if we want to change one of the generic rules?

# A nonprogramming example



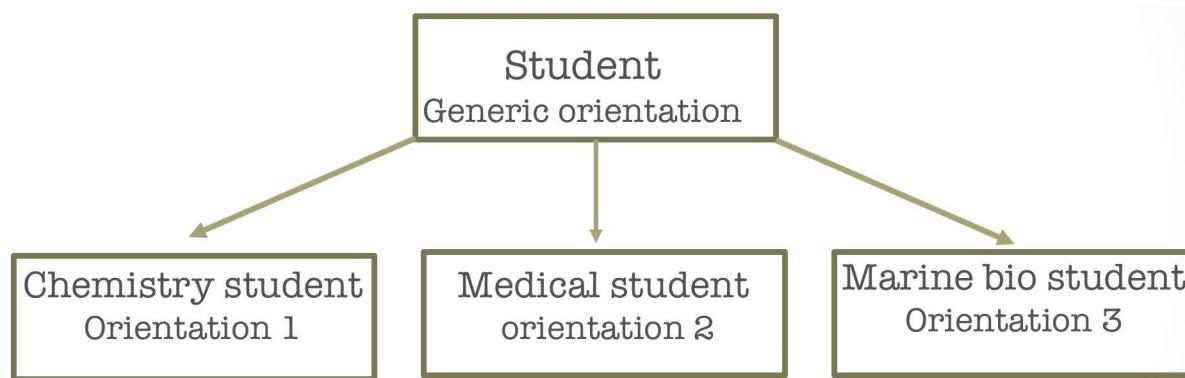
There is natural hierarchy

# Is-a relationship

- A hierarchical connection between two categories in which one types is a specialized version of the other.

# Is-a relationship

- A hierarchical connection between two categories in which one types is a specialized version of the other.

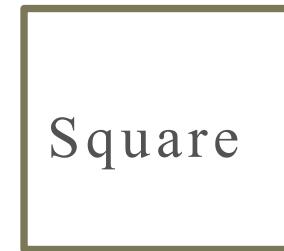


A chemistry student **is a** student

# Inheritance hierarchy

- A set of hierarchical relationships between classes of objects.

# Programming example



```
public class Triangle{  
    String color;  
    double area;  
}
```

```
public class Circle{  
    String color;  
    double area;  
}
```

```
public class Rectangle{  
    String color;  
    double area;  
}
```

```
public class Square{  
    String color;  
    double area;  
}
```

```
public class Triangle{ String  
    color; double area;
```

```
}
```

```
public class Circle{ String  
color; double area;
```

```
}
```

```
public class Rectangle{ String color;  
double area;  
}
```

```
public class Square{ String  
color; double area;  
}
```

what if I want to  
redefine color as  
an integer array (R,G,B)?

```
public class Triangle{ String color; double area;  
}
```

```
public class Circle{ String  
color; double area;  
}
```

```
public class Rectangle{ String color;  
double area;  
}
```

```
public class Square{ String  
color; double area;  
}
```

what if I want to  
redefine color as  
an integer array (R,G,B)?

What if I want to  
give each shape  
an outline color?

```
public class Triangle{ String color; double area;
```

```
}
```

```
public class Circle{ String  
color; double area;
```

```
}
```

```
public class Rectangle{ String color;  
double area;
```

```
}
```

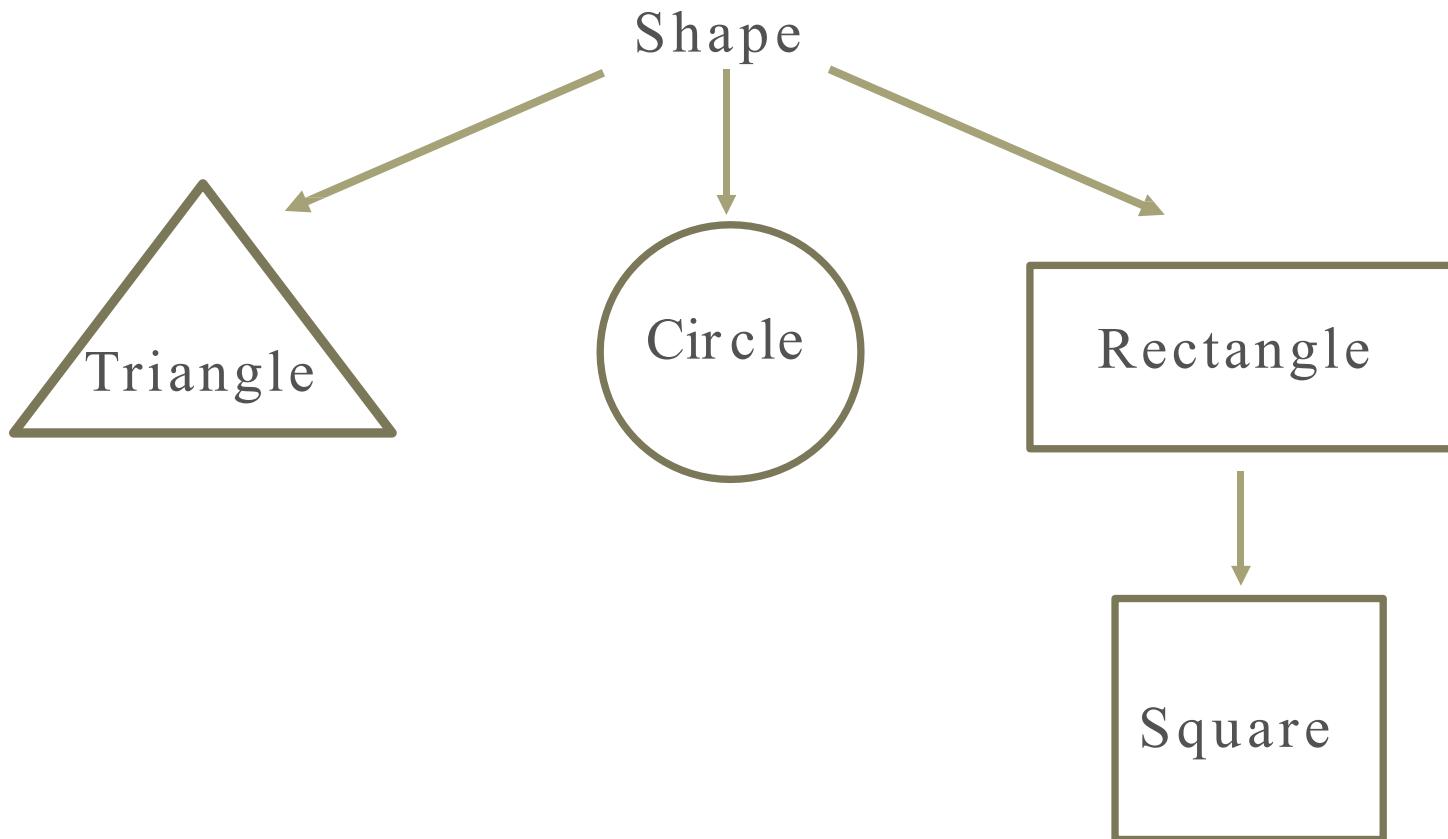
```
public class Square{ String  
color; double area;  
}
```

what if I want to  
redefine color as  
an integer array (R,G,B)?

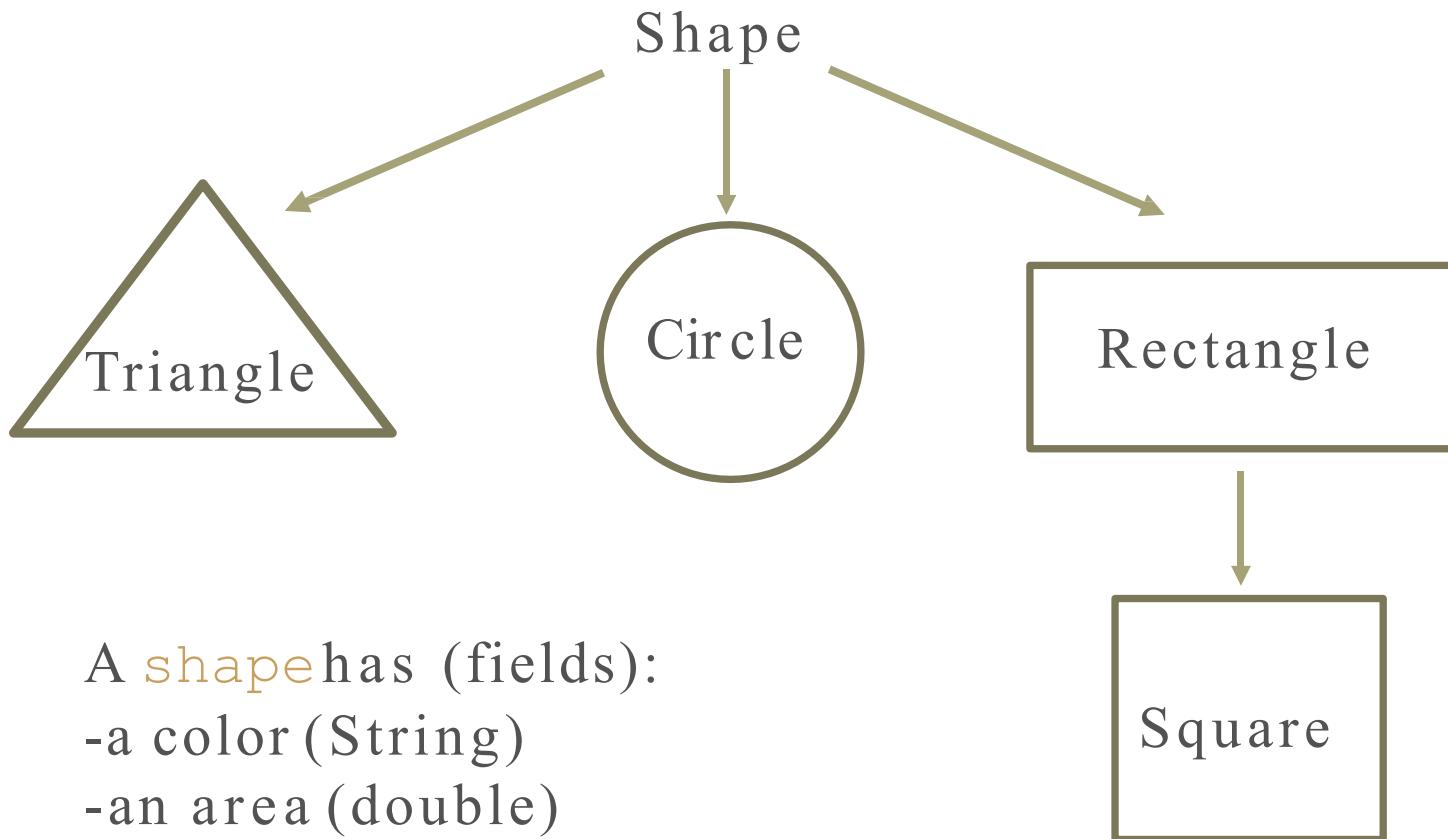
What if I want to  
give each shape  
an outline color?

**What can I do?**

# Inheritance hierarchy



# Shape



# Inheritance (inherit)

- A programming technique that allows a derived class to **extend** the functionality of a base class, inheriting all of its state and behavior

# Next Time...

- Inheritance, interfaces, generics
  - Reading
    - Chapter 9
    - Chapter 11
- Assignment 2 is up
  - Follow instructions exactly, otherwise, you'll lose points

# Quiz...

# Recursion

CSC220 | Computer Programming 2

Last Time...

# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.

# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

( )

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

( )

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

( )

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

[ ]

# Binary search

**ANOTHER EXAMPLE!**

( )

- What if I want to search (or sort) a collection of **objects**?

( )

# Comparator interface

```
public interface Comparator<T> {  
    int compare(T left, T right);  
}
```

- returns a number <0 if left < right
- returns a 0 if they are equal
- returns a number >0 if left > right

# How does it work?!

```
import java.util.Comparator;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(){
        super();
    }

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public double getArea(){
        return width*height;
    }

    protected class OrderByArea implements Comparator<Rectangle> {
        public int compare(Rectangle lhs, Rectangle rhs) {
            return (int) (lhs.getArea() - rhs.getArea());
        }
    }
}
```

Class type



```
1 // Java program to demonstrate accessing
2 // a inner class
3
4 // outer class
5 class OuterClass
6 {
7     // static member
8     static int outer_x = 10;
9
10    // instance(non-static) member
11    int outer_y = 20;
12
13    // private member
14    private int outer_private = 30;
15
16    // inner class
17    class InnerClass
18    {
19        void display()
20        {
21            // can access static member of outer class
22            System.out.println("outer_x = " + outer_x);
23
24            // can also access non-static member of outer class
25            System.out.println("outer_y = " + outer_y);
26
27            // can also access private member of outer class
28            System.out.println("outer_private = " + outer_private);
29
30        }
31    }
32 }
```

```
33
34 // Driver class
35 public class InnerClassDemo
36 {
37     public static void main(String[] args)
38     {
39         // accessing an inner class
40         OuterClass outerObject = new OuterClass();
41         OuterClass.InnerClass innerObject = outerObject.new InnerClass();
42
43         innerObject.display();
44     }
45 }
46
47
```

```
35 // Driver class
36 public class StaticNestedClassDemo
37 {
38     public static void main(String[] args)
39     {
40         // accessing a static nested class
41         OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
42
43         nestedObject.display();
44     }
45 }
46 }
47
1 // Java program to demonstrate
2 // a static nested class
3
4 // outer class
5 class OuterClass
6 {
7     // static member
8     static int outer_x = 10;
9
10    // instance(non-static) member
11    int outer_y = 20;
12
13    // private member
14    private static int outer_private = 30;
15
16    // static nested class
17    static class StaticNestedClass
18    {
19        void display()
20        {
21            // can access static member of outer class
22            System.out.println("outer_x = " + outer_x);
23
24            // can access display private static member of outer class
25            System.out.println("outer_private = " + outer_private);
26
27            // The following statement will give compilation error
28            // as static nested class cannot directly access non-static members
29            // System.out.println("outer_y = " + outer_y);
30
31        }
32    }
33 }
34 }
```

# Recursion

# Programming techniques

- Iterative approach
  - A programming approach in which you describe a **sequence** of actions.
- Recursion
  - A programming approach in which you describe actions be repeated using a method that calls **itself**.

# Recursion

- **Recursion** is a problem solving technique in which the solution is defined in terms of a **simpler (or smaller)** version of the problem
  - Break the problem into smaller parts

# sumexample

- Iterative approach

```
public static int sum(int n){  
    int sum = 0;  
  
    for (int i = 0; i < n; i++)  
        sum += i;  
    return sum;  
}
```

Today...

## sum example

- Some functions are easiest to define recursively

$$\text{sum}(N) = \text{sum}(N-1) + N$$

## sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)}_{\downarrow} + N$$

$$\text{sum}(N-1) = \text{sum}(N-2) + (N-1)$$

## sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underbrace{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$

⋮

?

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underbrace{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(1) \longrightarrow 1$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

$$\text{Sum}(1) \longrightarrow 1$$



# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

$$\text{Sum}(1) \longrightarrow 1$$



# Recursion

- A recursive method calls itself
- Some functions are easiest to define recursively

$$\text{sum}(N) = \text{sum}(N-1) + N$$

# Recursion

- A recursive method calls itself
  - Some functions are easiest to define recursively
    - $\text{sum}(N) = \text{sum}(N-1) + N$
  - There **must** be at least one *case* that can be computed without recursion
    - Any recursive call **must** make progress towards this case!
- $\longrightarrow \text{sum}(1) = 1$

# Recursion

- A recursive method calls itself → **recursive case**

- Some functions are easiest to define recursively

$$\text{sum}(N) = \text{sum}(N-1) + N$$

- There **must** be at least one *case* that can be computed without recursion

- Any recursive call **must** make progress towards this case! → **base case**

# sumexample

- Some functions are easiest to define recursively      recursive case

$$\text{sum}(N) = \boxed{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \text{sum}(N-2) + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

base case

$$\boxed{\text{Sum}(1)}$$

$$\longrightarrow 1$$



# sum example

- Recursive approach

$$\text{sum}(N) = \text{sum}(N-1) + N$$

```
public static int sum(int n) {  
}  
}
```

# sumexample

- Recursive approach

$$\text{sum}(N) = \text{sum}(N-1) + N$$

```
public static int sum(int n) {  
    recursive case  
    }  
    return sum(n-1) + n;
```

# sumexample

- Recursive approach

$$\text{sum}(N) = \text{sum}(N-1) + N$$

```
public static int sum(int n) {  
    base case if(n == 1)  
        return 1;  
    return sum(n-1) + n;  
}
```

# Recursion

- Recursion often seems like **MAGIC**
  - use this to your advantage

# Recursion

- Recursion often seems like **MAGIC**
  - use this to your advantage
- When writing a recursive method, just assume that the function you're writing already works, so you can use it to help solve the problem

# Recursion

- Recursion often seems like **MAGIC**
  - use this to your advantage
- When writing a recursive method, just assume that the function you're writing already works, so you can use it to help solve the problem
- Once you've worked out the recursion, think about the base case, and you're done

# Which is better?

- No clear answer, but there are tradeoffs
- Mathematicians often prefer recursive approach
  - Solutions are often shorter, closer in spirit to math
  - Good recursive solutions may be harder to design and test
- Programmers (college taught) often prefer iterative
  - Appealing to more people
  - Less “magical”, more in control

Factorial.....

- Let's try it....

# Factorial...

```
public static int factorial (int n) {  
    if (n==0) {  
        return 1;  
    } else {  
        return n * factorial (n-1);  
    }  
}
```

# Overhead of Recursion

# Methods calls

- Every time a method is invoked, a unique “frame” is created
  - Contains local variables and state
  - Put on the **call stack**



# Methods calls

- Every time a method is invoked, a unique “frame” is created
  - Contains local variables and state
  - Put on the **call stack**



# Methods calls

- Every time a method is invoked, a unique “frame” is created
  - Contains local variables and state
  - Put on the **call stack**
- When that method returns, execution resumes in the calling method



# Methods calls

- Every time a method is invoked, a unique “frame” is created
  - Contains local variables and state
  - Put on the **call stack**
- When that method returns, execution resumes in the calling method



This is how methods know where to return to!

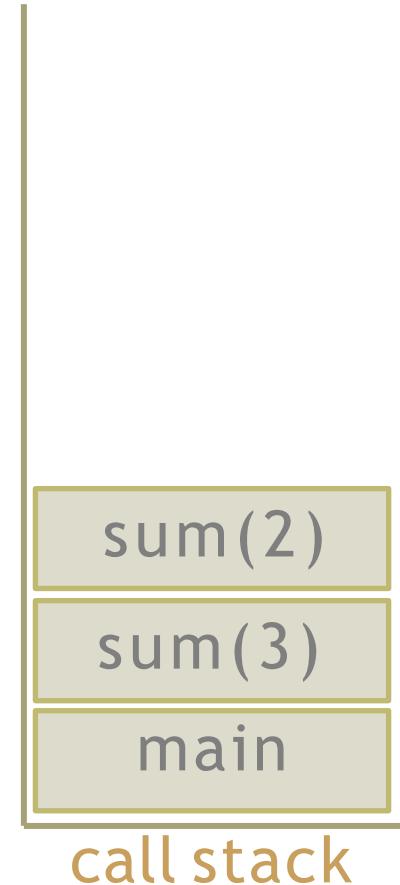
# Methods calls

- For recursive functions
  - Create multiple frames of the same method



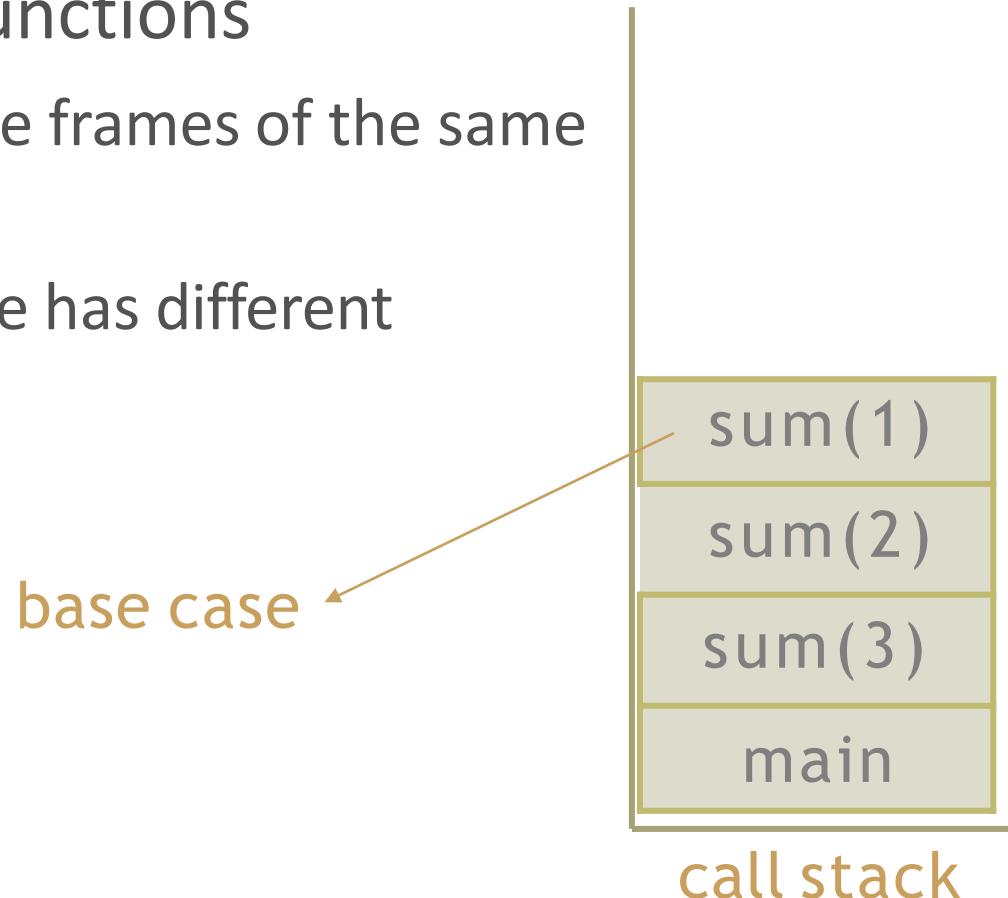
# Methods calls

- For recursive functions
  - Create multiple frames of the same method
  - But each frame has different arguments



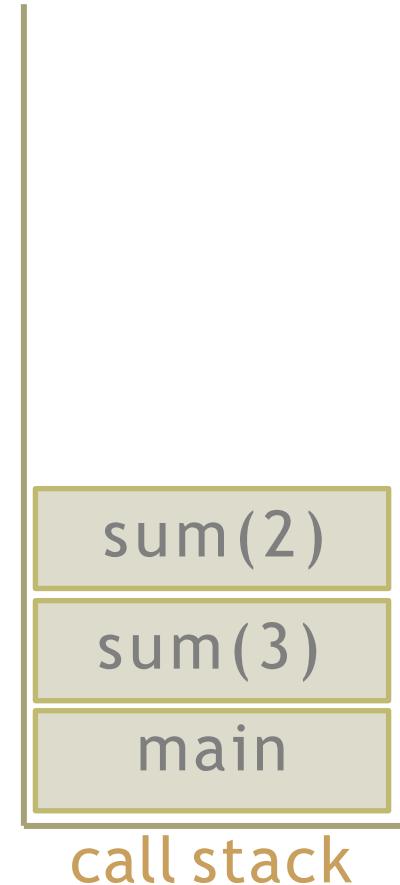
# Methods calls

- For recursive functions
  - Create multiple frames of the same method
  - But each frame has different arguments



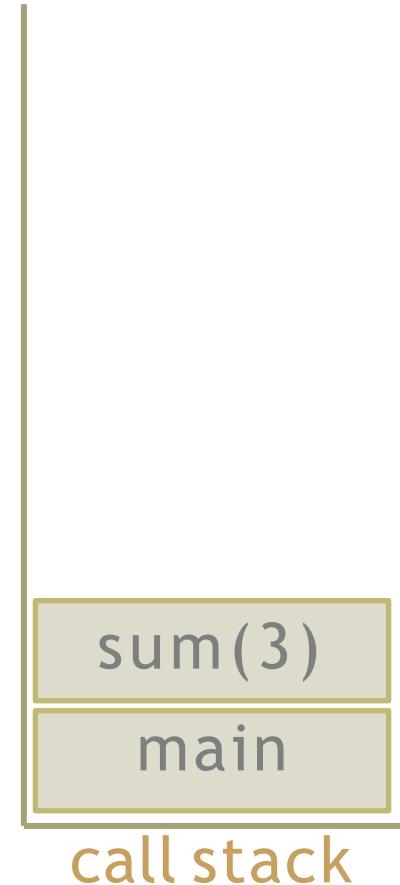
# Methods calls

- For recursive functions
  - Create multiple frames of the same method
  - But each frame has different arguments



# Methods calls

- For recursive functions
  - Create multiple frames of the same method
  - But each frame has different arguments



# Methods calls

- For recursive functions
  - Create multiple frames of the same method
  - But each frame has different arguments



# Four recursion rules

1. **Always** have at least one case that can be solved without using recursion
2. Any recursive call **must** progress toward a base case
3. **Always** assume that the recursive call works, and use this assumption to design your algorithms
4. **Never** duplicate work by solving the same instance of a problem in separate recursive calls

# Recursion caveat

- **Do not use recursion when a simple loop will do**
  - ...there is a lot of overhead involved in setting up the method frame
  - *Way more overhead than one iteration of a for-loop*

# Recursive binary search

- With the same signature

```
public static int binarySearch(int[] numbers, int target)
```

- How?!

# Helper function

```
public static int RecursiveBinarySearch(int[] numbers,  
                                      int target){  
    return binarySearchR(numbers, target, 0, numbers.length-1);  
}
```

---

```
// private recursive helper to implement binary search  
private static int binarySearchR(int[] numbers, int target,  
                                 int min, int max){  
    //...  
}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //base case

    //recursive case

}
```

# Recursive binary search

- **HINT**

- base case(s)?
- check if middle item is what we're looking for
  - *if so, return true*
- else, figure out if item is the left or right half
  - *repeat on that half*

# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //recursive case
    int mid = (max + min) /2;

}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //recursive case
    int mid = (max + min) /2;
    if (numbers[mid] == target){
        return mid;
    }

}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //recursive case
    int mid = (max + min) /2;
    if (numbers[mid] == target){
        return mid;
    }else if (numbers[mid] < target){

    }
}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //recursive case
    int mid = (max + min) /2;
    if (numbers[mid] == target){
        return mid;
    }else if (numbers[mid] < target){
        return binarySearchR(numbers, target, mid + 1, max);
    }
}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                int min, int max){

    //recursive case
    int mid = (max + min) /2;
    if (numbers[mid] == target){
        return mid;
    }else if (numbers[mid] < target){
        return binarySearchR(numbers, target, mid + 1, max);
    }else{
        return binarySearchR(numbers, target, min, mid-1);
    }
}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                 int min, int max){
    if (min > max){
        }else{
            //recursive case
            int mid = (max + min) /2;
            if (numbers[mid] == target){
                return mid;
            }else if (numbers[mid] < target){
                return binarySearchR(numbers, target, mid + 1, max);
            }else{
                return binarySearchR(numbers, target, min, mid-1);
            }
        }
}
```

# Recursive binary search

```
// private recursive helper to implement binary search
private static int binarySearchR(int[] numbers, int target,
                                 int min, int max){
    if (min > max){
        //base case
        return -1; // not found
    }else{
        //recursive case
        int mid = (max + min) /2;
        if (numbers[mid] == target){
            return mid;
        }else if (numbers[mid] < target){
            return binarySearchR(numbers, target, mid + 1, max);
        }else{
            return binarySearchR(numbers, target, min, mid-1);
        }
    }
}
```

Next Time...

- Analysis of Algorithms and sorting
  - Reading: Chapter 13
- Start early on your assignment
- Read the book before and after the class
- Practice coding

( )

# Searching & Recursion

## CSC220|Computer Programming 2

- Suppose we want to find an integer number in an array of ints.
- How can we do this?

# Sequential search

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

# Sequential search

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

- What if the array size was 10 million?

# Sequential search

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

- What if the array size was 10 million?
  - And the value we want was the last element

# Sequential search

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

- What if the array size was 10 million?
  - And the value we want was the last element

Can we do better?

# Example

- Finding a word in a dictionary
  - 1. Guess which page the entry is on
  - 2. Did we go too far?
    - Go back some pages
  - 3. Did we not go far enough?
    - Go forward some pages
  - 4. Continue narrowing

# Example

- Finding a word in a dictionary

1. Guess which page the entry is on

2. Did we go too far?

- Go back some pages

3. Did we not go far enough?

- Go forward some pages

4. Continue narrowing

what does this algorithm assume  
about the dictionary?

- If we are searching through elements of an array or list that we **know** is in sorted order.
- We can use a better algorithm called **binary search** instead of sequential search

# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.

# Binary search

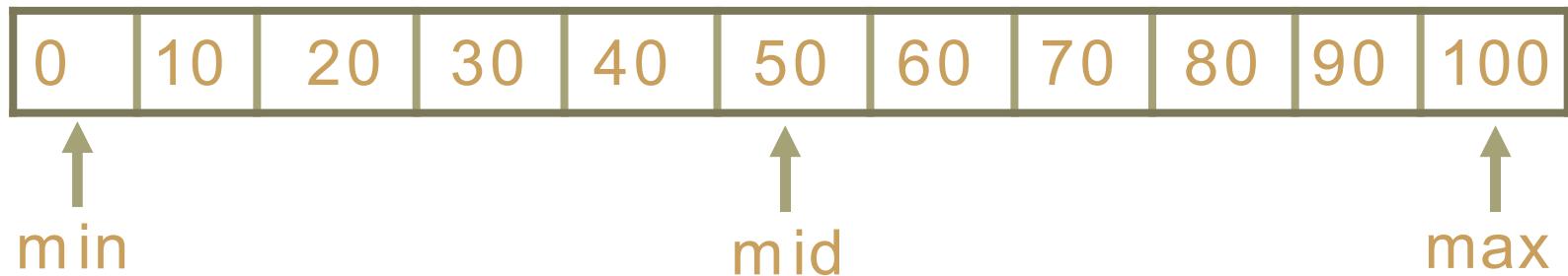
- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
- How does it work?

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

Searching for 80...

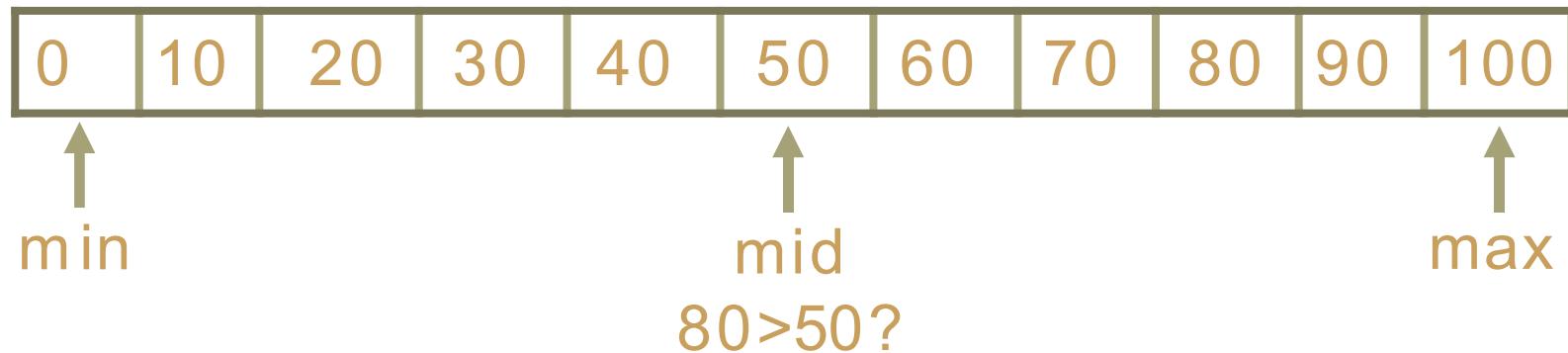
# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
- How does it work?



# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
  - How does it work?



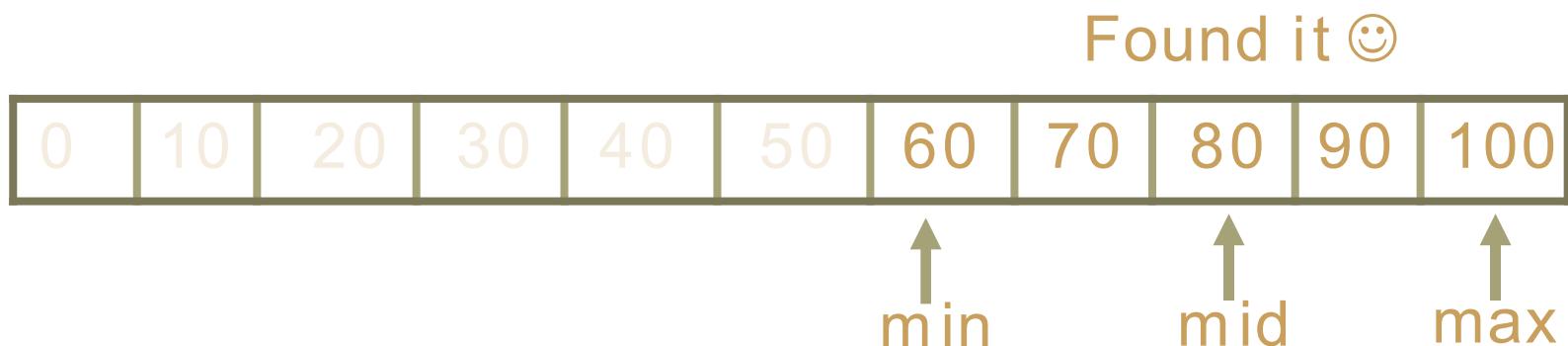
# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
- How does it work?



# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
- How does it work?



# Binary search

- An algorithm that searches for a value in a sorted list by repeatedly diving the search space **in half**.
- How does it work?

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

Scales extremely well to large data!

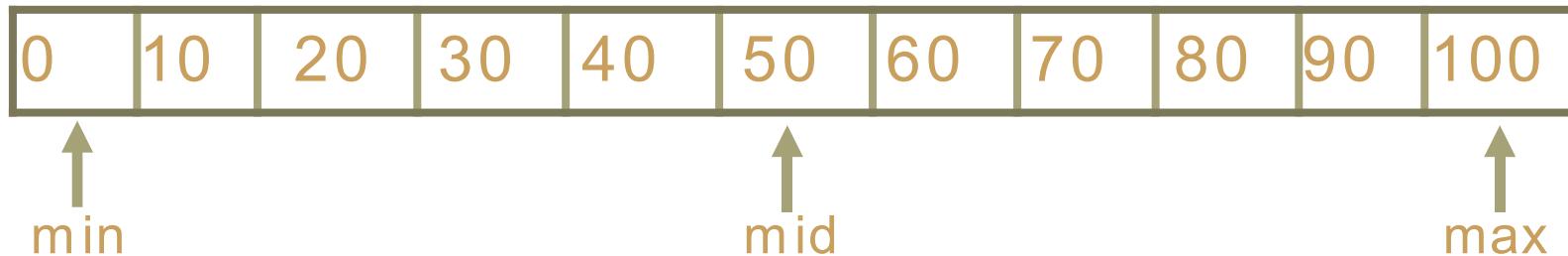
# Binary search

- We must keep track of three indexes:
  - The maximum index of interest (max)
  - The minimum index of interest (min)
  - The middle index (halfway through min and max)

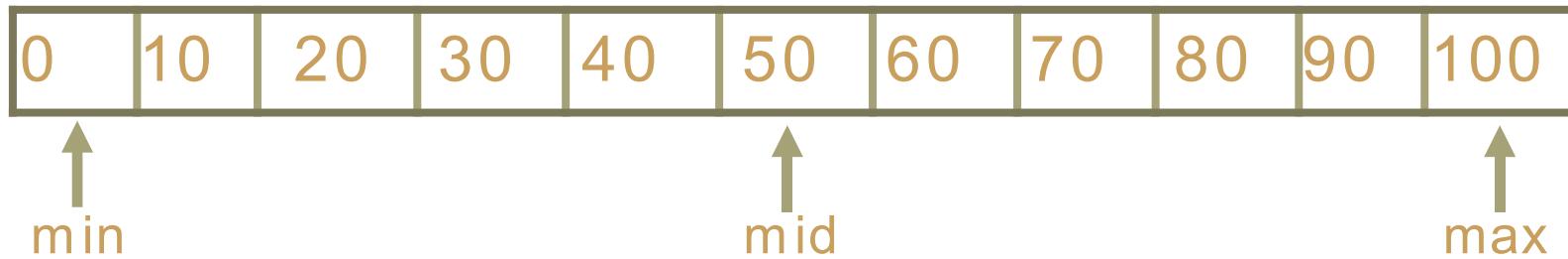
# Binary search

- We must keep track of three indexes:
  - The maximum index of interest (max)
  - The minimum index of interest (min)
  - The middle index (halfway through min and max)
- Any special case?
  - What if the value is not in the array?
    - Let's say we are interested in finding 75...

# Binary search



# Binary search



# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max

# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max

Can I continue?!

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

- What if I want to search (or sort) a collection of **objects**?

# Sequential search for objects

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

# Sequential search for objects

```
public static int FindIndex(ArrayList<Integer> array, int val){  
    int index = -1;  
    for (int i = 0; i < array.size(); i++){  
        if (array.get(i) == val){  
            return i;  
        }  
    }  
  
    return index;  
}
```

# Sequential search for object

```
public int indexOf(LibraryBook[] books, LibraryBook target){  
    for (int i = 0; i < books.length; i++){  
        if (books[i].equals(target)){  
            return i;  
        }  
    }  
    return -1; //not found  
}
```

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

How to compare objects?!

# Comparable interface

- `java.lang.Comparable` interface has one method:
  - `compareTo()`
    - *takes one argument*
    - *decides which one is greater*
- Classes that implement Comparable have a natural ordering
  - *Can be sorted without knowing any details about the class*
  - *just use the compareTo() method!*

# Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T item);  
}
```

- defines a natural ordering (*in fact, it is contractually obligated to!*)
  - String, Integer, ... all implement Comparable

# Binary search

```
public static int BinarySearch(String[] strings, String target){  
    int min = 0;  
    int max = strings.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        int compare = strings[mid].compareTo(target);  
        if (compare == 0){  
            return mid; // found it!  
        }else if (compare < 0){  
            min = mid + 1; // too small  
        }else{ // compare > 0  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

# Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T item);  
}
```

- defines a natural ordering (*in fact, it is contractually obligated to!*)
  - String, Integer, ... all implement Comparable
- What if we want a different ordering? or to order Shapes or LibraryBooks? or to order Strings based on length?

# Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T item);  
}
```

- defines a natural ordering (*in fact, it is contractually obligated to!*)
  - String, Integer, ... all implement Comparable
- What if we want a different ordering? or to order Shapes or LibraryBooks? or to order Strings based on length?

—————> Need to implement your own comparison function

# Comparator interface

```
public interface Comparator<T> {  
    int compare(T left, T right);  
}
```

- returns a number <0 if left < right
- returns a 0 if they are equal
- returns a number >0 if left > right

# How does it work?!

```
import java.util.Comparator;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(){
        super();
    }

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public double getArea(){
        return width*height;
    }

    protected class OrderByArea implements Comparator<Rectangle> {
        public int compare(Rectangle lhs, Rectangle rhs) {
            return (int) (lhs.getArea() - rhs.getArea());
        }
    }
}
```

# How does it work?!

```
import java.util.Comparator;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(){
        super();
    }

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public double getArea(){
        return width*height;
    }

    protected class OrderByArea implements Comparator<Rectangle> {
        public int compare(Rectangle lhs, Rectangle rhs) {
            return (int) (lhs.getArea() - rhs.getArea());
        }
    }
}
```

# How does it work?!

```
import java.util.Comparator;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(){
        super();
    }

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public double getArea(){
        return width*height;
    }

    protected class OrderByArea implements Comparator<Rectangle> {
        public int compare(Rectangle lhs, Rectangle rhs) {
            return (int) (lhs.getArea() - rhs.getArea());
        }
    }
}
```

Class type

# How does it work?!

```
import java.util.Comparator;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(){
        super();
    }

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public double getArea(){
        return width*height;
    }

    protected class OrderByArea implements Comparator<Rectangle> {
        public int compare(Rectangle lhs, Rectangle rhs) {
            return (int) (lhs.getArea() - rhs.getArea());
        }
    }
}
```

Class type

# How/when to use it?!

You will see in tomorrow's lab!

```
1 // A Java program to demonstrate use of Comparable
2 import java.io.*;
3 import java.util.*;
4
5 // A class 'Movie' that implements Comparable
6 class Movie implements Comparable<Movie>
7 {
8     private double rating;
9     private String name;
10    private int year;
11
12    // Used to sort movies by year
13    public int compareTo(Movie m)
14    {
15        return this.year - m.year;
16    }
17
18    // Constructor
19    public Movie(String nm, double rt, int yr)
20    {
21        this.name = nm;
22        this.rating = rt;
23        this.year = yr;
24    }
25
26    // Getter methods for accessing private data
27    public double getRating() { return rating; }
28    public String getName() { return name; }
29    public int getYear() { return year; }
30 }
31
32 // Driver class
33 class Main
34 {
35     public static void main(String[] args)
36     {
37         ArrayList<Movie> list = new ArrayList<Movie>();
38         list.add(new Movie("Force Awakens", 8.3, 2015));
39         list.add(new Movie("Star Wars", 8.7, 1977));
40         list.add(new Movie("Empire Strikes Back", 8.8, 1980));
41         list.add(new Movie("Return of the Jedi", 8.4, 1983));
42
43         Collections.sort(list);
44
45         System.out.println("Movies after sorting : ");
46         for (Movie movie: list)
47         {
48             System.out.println(movie.getName() + " " +
49                             movie.getRating() + " " +
50                             movie.getYear());
51         }
52     }
53 }
```

```
1 //A Java program to demonstrate Comparator interface
2 import java.io.*;
3 import java.util.*;
4
5 // A class 'Movie' that implements Comparable
6 class Movie implements Comparable<Movie>
7 {
8     private double rating;
9     private String name;
10    private int year;
11
12    // Used to sort movies by year
13    public int compareTo(Movie m)
14    {
15        return this.year - m.year;
16    }
17
18    // Constructor
19    public Movie(String nm, double rt, int yr)
20    {
21        this.name = nm;
22        this.rating = rt;
23        this.year = yr;
24    }
25
26    // Getter methods for accessing private data
27    public double getRating() { return rating; }
28    public String getName() { return name; }
29    public int getYear() { return year; }
30 }
31
32 // Class to compare Movies by ratings
33 class RatingCompare implements Comparator<Movie>
34 {
35     public int compare(Movie m1, Movie m2)
36     {
37         if (m1.getRating() < m2.getRating()) return -1;
38         if (m1.getRating() > m2.getRating()) return 1;
39         else return 0;
40     }
41 }
42
43 // Class to compare Movies by name
44 class NameCompare implements Comparator<Movie>
45 {
46     public int compare(Movie m1, Movie m2)
47     {
48         return m1.getName().compareTo(m2.getName());
49     }
50 }
51
52 // Driver class
53 class Main
54 {
55     public static void main(String[] args)
56     {
57         ArrayList<Movie> list = new ArrayList<Movie>();
58         list.add(new Movie("Force Awakens", 8.3, 2015));
59         list.add(new Movie("Star Wars", 8.7, 1977));
60         list.add(new Movie("Empire Strikes Back", 8.8, 1980));
61         list.add(new Movie("Return of the Jedi", 8.4, 1983));
62
63         // Sort by rating : (1) Create an object of ratingCompare
64         //                      (2) Call Collections.sort
65         //                      (3) Print Sorted list
66         System.out.println("Sorted by rating");
67         RatingCompare ratingCompare = new RatingCompare();
68         Collections.sort(list, ratingCompare);
69         for (Movie movie: list)
70             System.out.println(movie.getRating() + " " +
71                               movie.getName() + " " +
72                               movie.getYear());
```

```
74
75     // Call overloaded sort method with RatingCompare
76     // (Same three steps as above)
77     System.out.println("\nSorted by name");
78     NameCompare nameCompare = new NameCompare();
79     Collections.sort(list, nameCompare);
80     for (Movie movie: list)
81         System.out.println(movie.getName() + " " +
82                           movie.getRating() + " " +
83                           movie.getYear());
84
85     // Uses Comparable to sort by year
86     System.out.println("\nSorted by year");
87     Collections.sort(list);
88     for (Movie movie: list)
89         System.out.println(movie.getYear() + " " +
90                           movie.getRating() + " " +
91                           movie.getName());
```

92 }

93 }

- Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
- Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
- If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on their natural order defined by compareTo method.

***To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.***

# Recursion

# Programming techniques

- Iterative approach
  - A programming approach in which you describe a sequence of actions.

# Programming techniques

- Iterative approach
  - A programming approach in which you describe a **sequence** of actions.
- Recursion
  - A programming approach in which you describe actions be repeated using a method that calls **itself**.

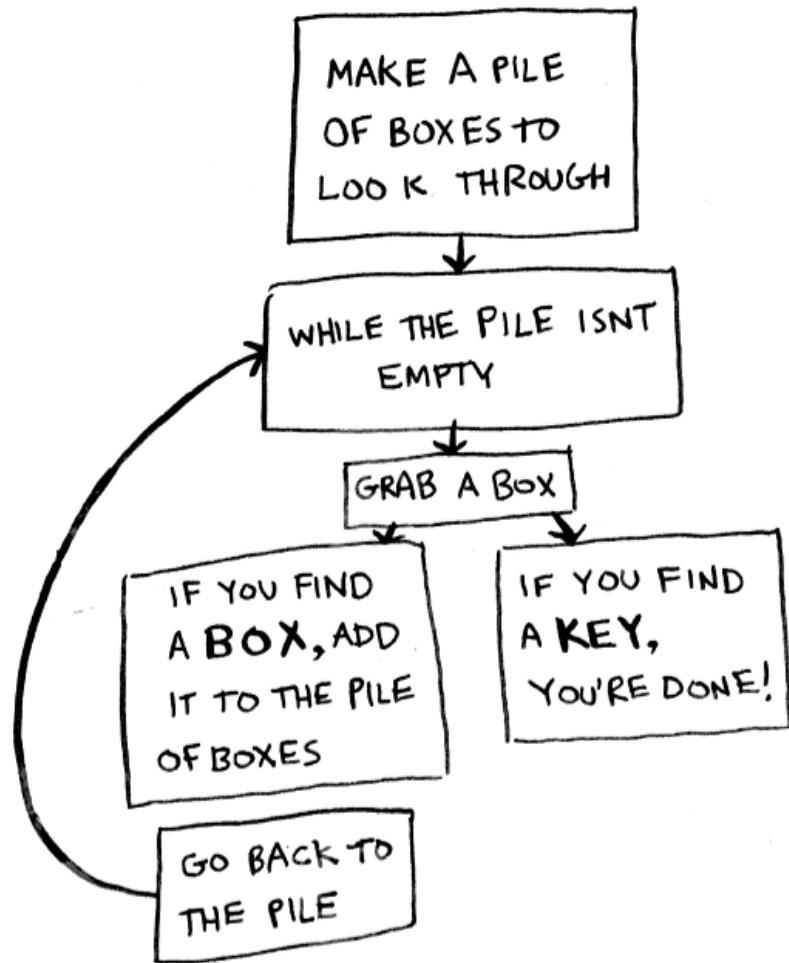
# Recursion

- **Recursion** is a problem solving technique in which the solution is defined in terms of a **simpler (or smaller)** version of the problem
  - Break the problem into smaller parts

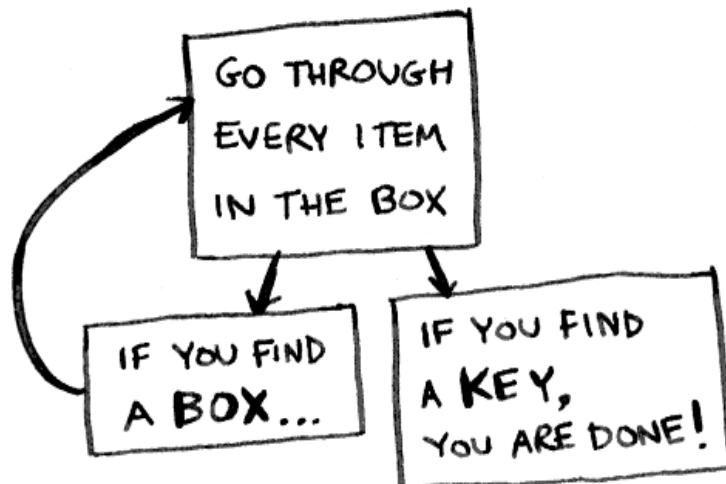


# Recursion

## Iterative Approach



## Recursive Approach



# sum example

- Iterative approach

```
public static int sum(int n){  
    int sum = 0;  
  
    for (int i = 0; i < n; i++)  
        sum += i;  
    return sum;  
}
```

# sum example

- Some functions are easiest to define recursively

$$\text{sum}(N) = \text{sum}(N-1) + N$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \text{sum}(N-2) + (N-1)$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underbrace{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



?

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underbrace{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underbrace{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(1) \longrightarrow 1$$

# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

$$\text{Sum}(1) \longrightarrow 1$$



# sumexample

- Some functions are easiest to define recursively

$$\text{sum}(N) = \underline{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \underline{\text{sum}(N-2)} + (N-1)$$



$$\text{sum}(N-2) = \underline{\text{sum}(N-3)} + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

$$\text{Sum}(1) \longrightarrow 1$$



# Recursion

- A recursive method calls itself
- Some functions are easiest to define recursively

$$\text{sum}(N) = \text{sum}(N-1) + N$$

- There must be at least one *case* that can be computed without recursion
  - Any recursive call must make progress towards this case!

# Recursion

- A recursive method calls itself → **recursive case**
- Some functions are easiest to define recursively
  - $\text{sum}(N) = \text{sum}(N-1) + N$
- There **must** be at least one *case* that can be computed without recursion
- Any recursive call **must** make progress towards this case! → **base case**

# sumexample

- Some functions are easiest to define recursively
- recursive case

$$\text{sum}(N) = \boxed{\text{sum}(N-1)} + N$$



$$\text{sum}(N-1) = \text{sum}(N-2) + (N-1)$$



$$\text{sum}(N-2) = \text{sum}(N-3) + (N-2)$$



$$\text{Sum}(2) = \text{sum}(1) + 2$$

base case

$$\boxed{\text{Sum}(1)}$$

$$\longrightarrow 1$$



# sumexample

- Recursive approach

$$\text{sum}(N) = \text{sum}(N-1) + N$$

```
public static int sum(int n) {  
    base case      if(n == 1)  
                    return 1;  
    return sum(n-1) + n;  
}
```

# sumexample

- Recursive approach

$$\text{sum}(N) = \text{sum}(N-1) + N$$

recursive case

```
public static int sum(int n) {  
    if(n == 1)  
        return 1;  
    return sum(n-1) + n;  
}
```

# Next Time...

- **quiz** on Thursday

- From generic programming and searching
- You have a lab tomorrow

# Analysis of Algorithm

## CSC220|Computer Programming 2

Correctness is only half the  
battle!!!

- Correctness is only half the battle
- Programs are expected to terminate in a reasonable amount of time

- Correctness is only half the battle
- Programs are expected to terminate in a reasonable amount of time
- Running time of a program is strongly correlated to the choice of algorithms used in problem solving

How much time and space does an algorithm require?

# Example

- Finding a word in a dictionary
- Algorithm 1
  - 1. Start on the **first** page, **first** entry
  - 2. If word not found, move to the next entry
  - 3. If very end of dictionary reached, word not found

# Example

- Finding a word in a dictionary
- Algorithm 1
  - 1. Start on the **first** page, **first** entry
  - 2. If word not found, move to the next entry
  - 3. If very end of dictionary reached, word not found

Does this work?  
Can we do any better?

# Example

- Finding a word in a dictionary
- Algorithm 2
  1. Guess which page the entry is on
  2. Did we go too far?
    - Go back some pages
  3. Did we not go far enough?
    - Go forward some pages
  4. Continue narrowing

what does this algorithm assume about the dictionary?

- Algorithm 1
  - Run time directly related to size of dictionary
    - Assume 100k words, each take .25sec → ~7hours!
- Algorithm 2
  - More like what humans do
  - A matter of seconds...

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

$1 \times 10^{12} \text{ *(minuscule amount of time) = large amount of time}$

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

$1 \times 10^{12} * (\text{minuscule amount of time}) = \text{large amount of time}$

- That is why algorithm **complexity matters!**

# Complexity

- A measure of the computing resources that are used by a piece of code
  - Time
  - Memory
  - Disk space

# Complexity

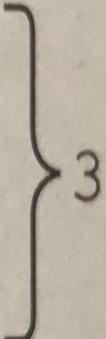
- Empirical analysis: timing how long it takes for a program to run
- Not very useful because different computers can run at different speeds
- Algorithm analysis is what we are after
  - The practice of applying techniques to mathematically approximate the performance of various computing algorithms

# N

- N is the problem size
  - *Sorting an array of N numbers*
  - *Searching for an item in a set of N items*
  - *Inserting an item into a set of N items*
- Amount of work done for these operations usually depends on N
  - Work required is a **function** of N
- Algorithms don't always require N steps for N items!

# Complexity

```
statement1.  
statement2.  
statement3.
```



3

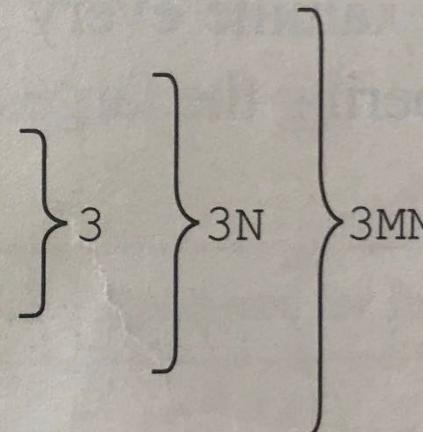
# Complexity

```
statement1.  
for (N times) { }  
    statement2. } N  
}  
  
for (M times) { }  
    statement3. } M  
}  
statement4.  
statement5.
```

The diagram illustrates the time complexity of a nested loop structure. It shows a sequence of statements: 'statement1.', followed by a 'for' loop that executes 'N' times, containing 'statement2.'. Below this is another 'for' loop that executes 'M' times, containing 'statement3.'. After these loops, there are two more statements: 'statement4.' and 'statement5.'. Braces on the right side group the code into three main sections: the single statement 'statement1.', the nested block of the first 'for' loop ('statement2.' and its brace), and the entire block of the second 'for' loop ('statement3.' and its brace). A large brace on the far right groups all these sections together, with the expression 'M + N + 3' written next to it, representing the total number of operations.

# Complexity

```
for (M times) {  
    for (N times) {  
        statement1.  
        statement2.  
        statement3.  
    }  
}
```



# Complexity

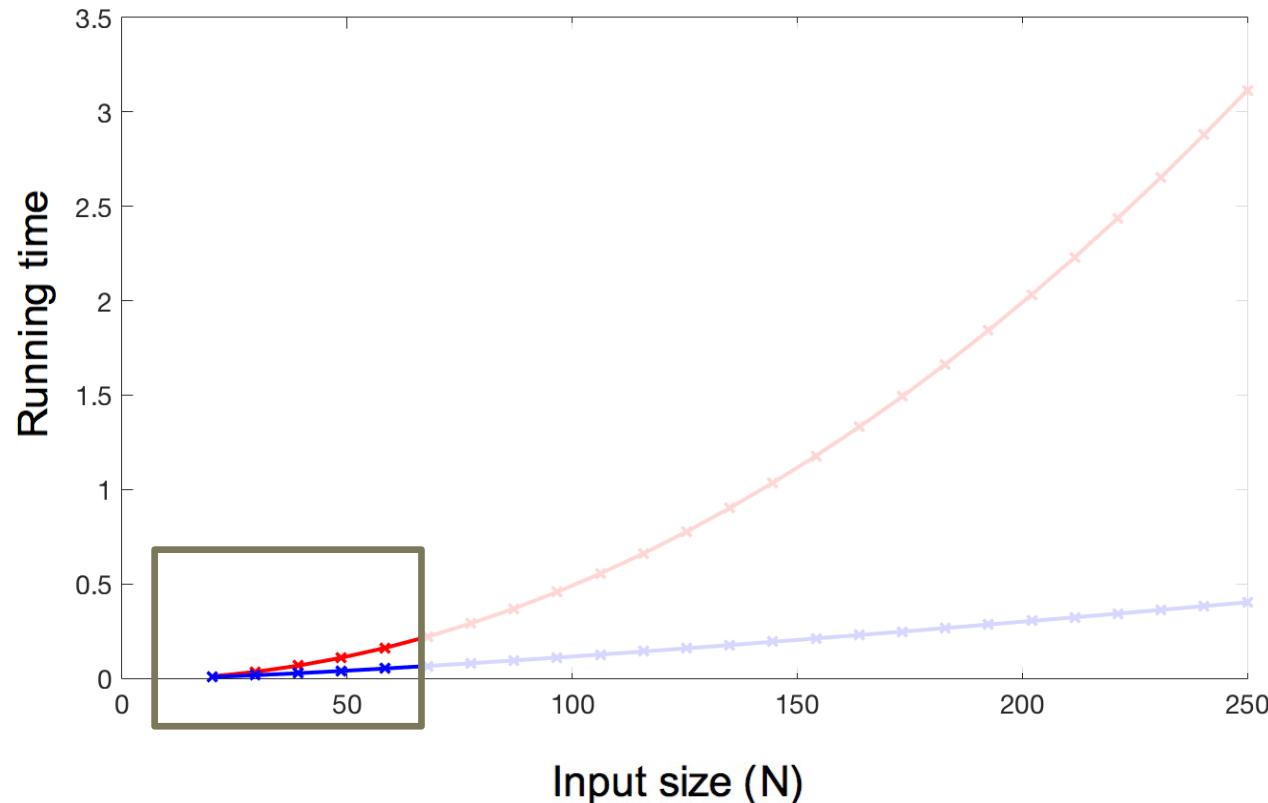
```
statement1.  
for (N times) {      }  
              statement2.  
    }                      } N  
  
for (M times) {      }  
              statement3.  
    }                      } M  
statement4.  
statement5.
```

The diagram illustrates the execution flow of the provided pseudocode. It features two nested loops: an outer loop (M times) containing an inner loop (N times). The code consists of five statements: statement1, statement2, statement3, statement4, and statement5. Statement1 is executed once. Statement2 is part of the inner loop and is executed  $N$  times. Statement3 is part of the outer loop and is executed  $M$  times. Statement4 and Statement5 are executed once each. Braces on the right side of the code group the loops and their associated statements. A large brace groups all statements except statement1, indicating its execution count. Another brace groups the inner loop's statements (statement2 and statement3), indicating its execution count relative to the outer loop. The total complexity is labeled as  $M + N + 3$ .

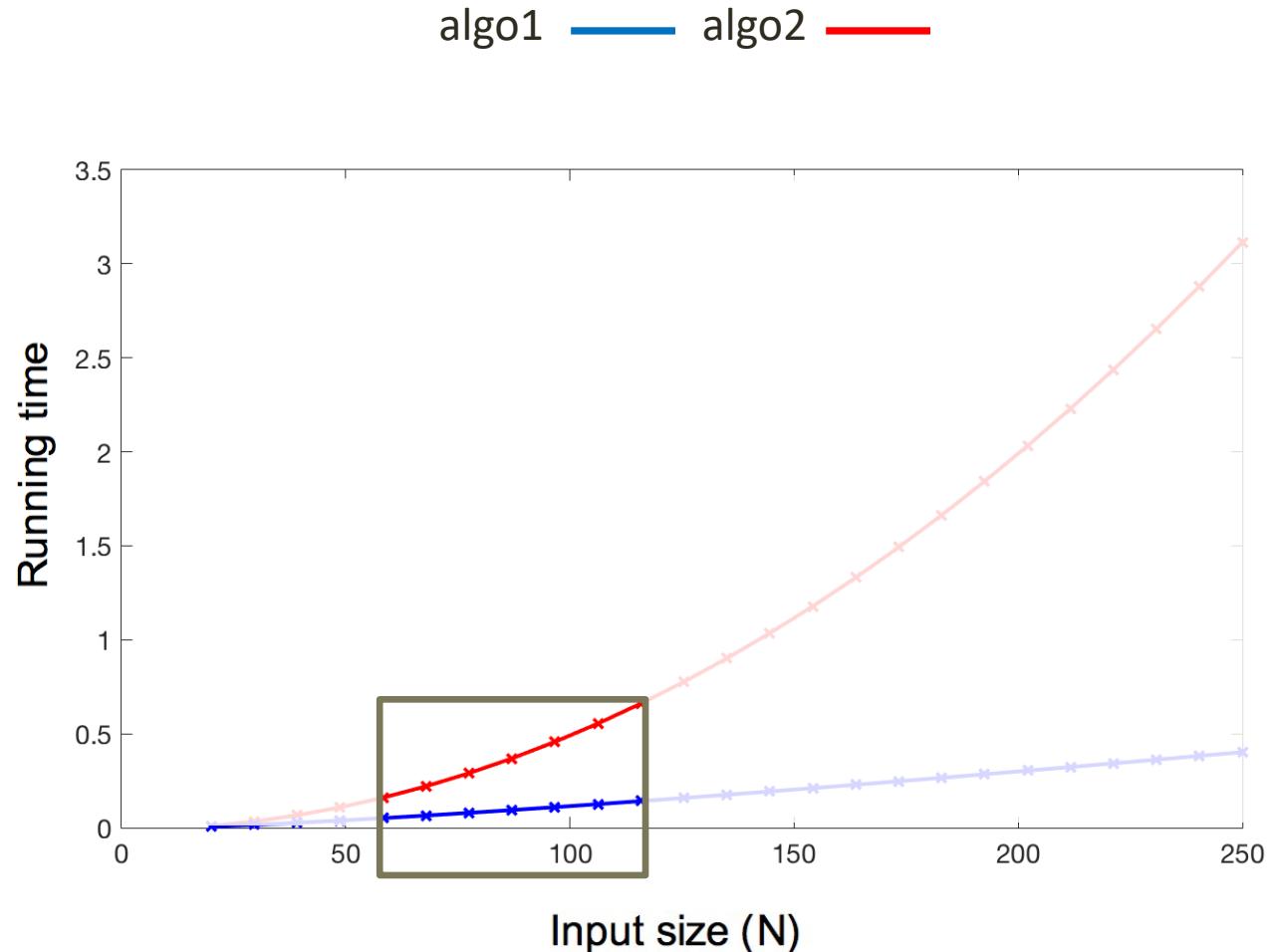
# Example

# small $N$

algo1 — algo2 —

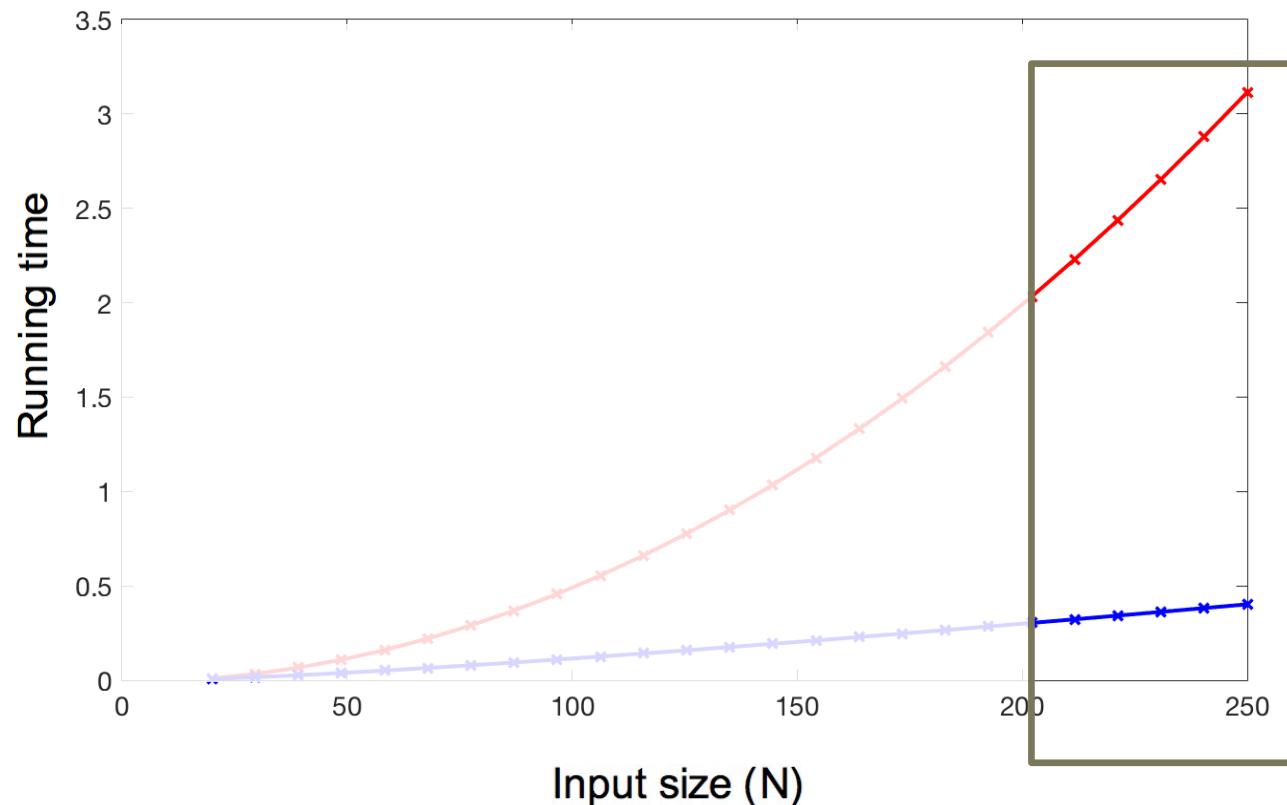


# medium N



# large N

algo1 — algo2 —



When **N** becomes large complexity  
matters!

# Growth Rate

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates

# Growth Rate

|            |              |
|------------|--------------|
| C          | constant     |
| $\log N$   | logarithmic  |
| $N$        | linear       |
| $N \log N$ | linearithmic |
| $N^2$      | quadratic    |
| $N^3$      | cubic        |

# Growth Rate

|            |              |
|------------|--------------|
| $C$        | constant     |
| $\log N$   | logarithmic  |
| $N$        | linear       |
| $N \log N$ | linearithmic |
| $N^2$      | quadratic    |
| $N^3$      | cubic        |

↓

Increasing growth rate

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates
- for sufficiently large  $N$ , a function's growth rate is determined by its dominate term

$N^2 + 1000N + 500$  —————> What is the dominate term?

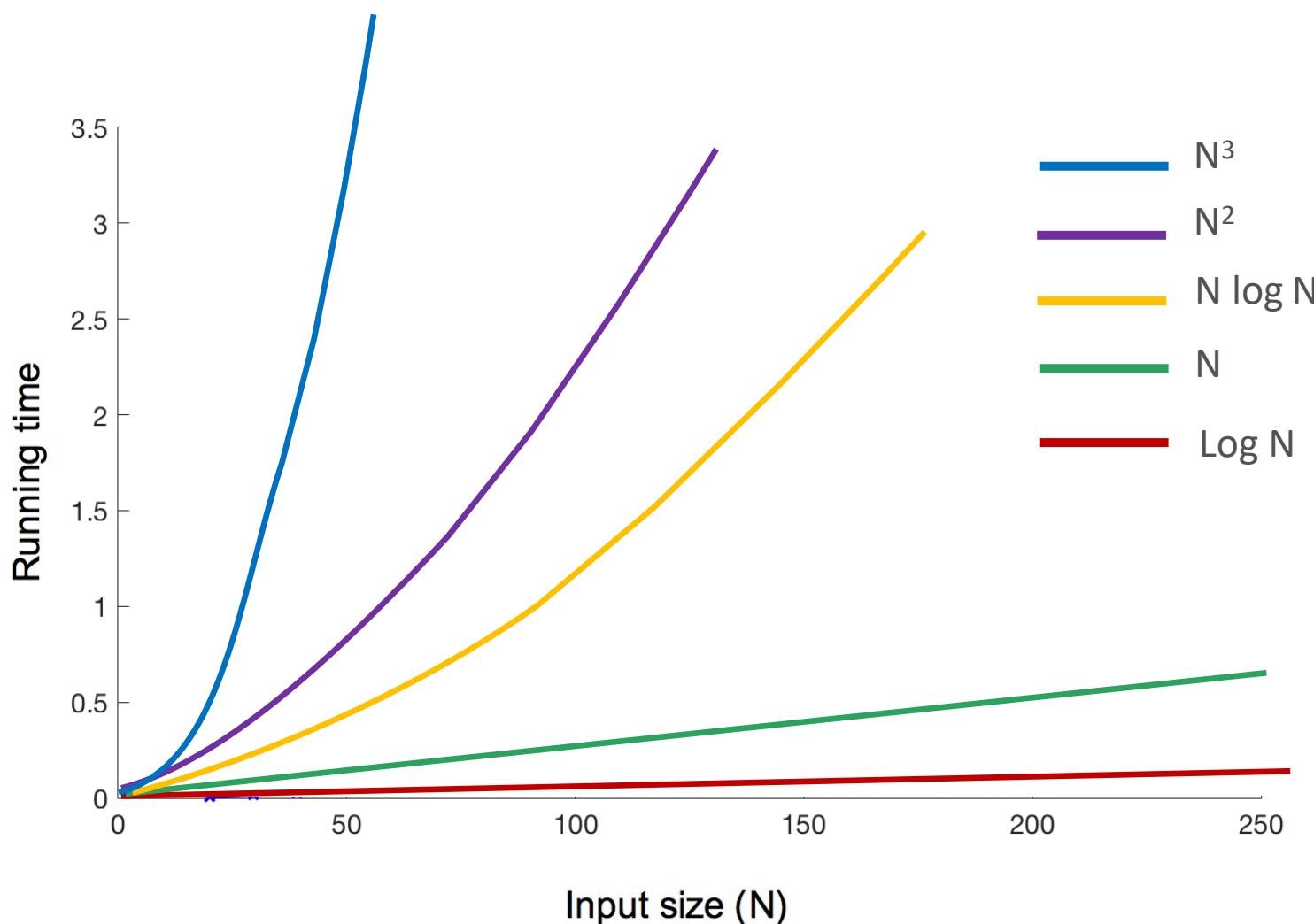
- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates
- for sufficiently large  $N$ , a function's growth rate is determined by its dominate term

|            |              |                                                                                                                |
|------------|--------------|----------------------------------------------------------------------------------------------------------------|
| $C$        | constant     | <br>Increasing growth rate |
| $\log N$   | logarithmic  |                                                                                                                |
| $N$        | linear       |                                                                                                                |
| $N \log N$ | linearithmic |                                                                                                                |
| $N^2$      | quadratic    |                                                                                                                |
| $N^3$      | cubic        |                                                                                                                |

# Big-O Notation

- **Big-O notation** ( $O$ ) is used to capture the dominate term in an algorithm
  - assuming large  $N$
- for example, the running time of a quadratic algorithm is  $N^2$  is specified  $O(N^2)$ 
  - read “order  $N$  squared”
- this notation allows us to establish a relative order among algorithms
  - $O(N \log N)$  is better than  $O(N^2)$

# Typical run-time complexities



- Constant -  $O(1)$ : have runtimes that don't depend on input size. Ex. Code to convert fahrenheit to celcius
- Logarithmic -  $O(\log N)$ : divide a problem space in half. Ex. Binary search
- Linear –  $O(N)$ : algorithms that are directly proportional to  $N$ . Ex. Counting, summing over a list of  $N$  items.

- Log Linear-  $O(N \log N)$ : have a combo of linear and log. Ex. Merge sort
- Quadratic-  $O(N^2)$ : runtimes proportional to the square of input size. Ex: double *for* loops
- Cubic–  $O(N^3)$ : runtimes proportional to the cube of input size. Ex. Triple *for* loops (3-d arrays)
- Exponential –  $O(2^N)$ : runtimes proportional to 2 raised to the power of N. EX: yeah, don't do these unless it's a small N.

# Example

- Finding the maximum item in an array
  - Initialize max to the first element
  - Scan through each item in the array
    - If the item is greater than max, update max

What is the big-o complexity of this algorithm?

- 1.c
- 2.log N
- 3.N
- 4.N log N
- 5.N<sup>2</sup>
- 6.N<sup>3</sup>

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;  
for(int i=0; i<array.length-1; i++) {  
    int num1 = array[i];  
    for(int j=i+1; j<array.length; j++) {  
        int num2 = array[j];  
        if (Math.abs(num1-num2) < diff)  
            diff = Math.abs(num1-num2);  
    }  
}  
  
return diff;
```

What is the big-o complexity of this algorithm?

c  
 $\log N$   
 $N$   
 $N \log N$   
 $N^2$   
 $N^3$

**O(n<sup>2</sup>)**

```
1 // C++ program to find minimum difference between
2 // any pair in an unsorted array
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 // Returns minimum difference between any pair
7 int findMinDiff(int arr[], int n)
8 {
9     // Sort array in non-decreasing order
10    sort(arr, arr+n);
11
12    // Initialize difference as infinite
13    int diff = INT_MAX;
14
15    // Find the min diff by comparing adjacent
16    // pairs in sorted array
17    for (int i=0; i<n-1; i++)
18        if (arr[i+1] - arr[i] < diff)
19            diff = arr[i+1] - arr[i];
20
21    // Return min diff
22    return diff;
23}
24
25 // Driver code
26 int main()
27 {
28     int arr[] = {1, 5, 3, 19, 18, 25};
29     int n = sizeof(arr)/sizeof(arr[0]);
30     cout << "Minimum difference is " << findMinDiff(arr, n);
31     return 0;
32}
```

## Method 2 (Efficient: O(n Log n))

The idea is to use sorting. Below are steps.

- 1) Sort array in ascending order. This step takes  $O(n \log n)$  time.
- 2) Initialize difference as infinite. This step takes  $O(1)$  time.
- 3) Compare all adjacent pairs in sorted array and keep track of minimum difference. This step takes  $O(n)$  time.

# Analyze the running time

```
for(int i=0; i<n; i+=2)  
    sum++;
```

1.c

```
for(int i=0; i<n; i++)  
    for(int j=0; j<n*n; j++)  
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

# Analyze the running time

```
for(int i=0; i<n; i+=2)
    sum++;
```

1.c

```
for(int i=0; i<n; i++)
    for(int j=0; j<n*n; j++)
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

# Analyze the running time

```
for(int i=0; i<n; i+=2)  
    sum++;
```

1.c

```
for(int i=0; i<n; i++)  
    for(int j=0; j<n*n; j++)  
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

What is the complexity of binary search?

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

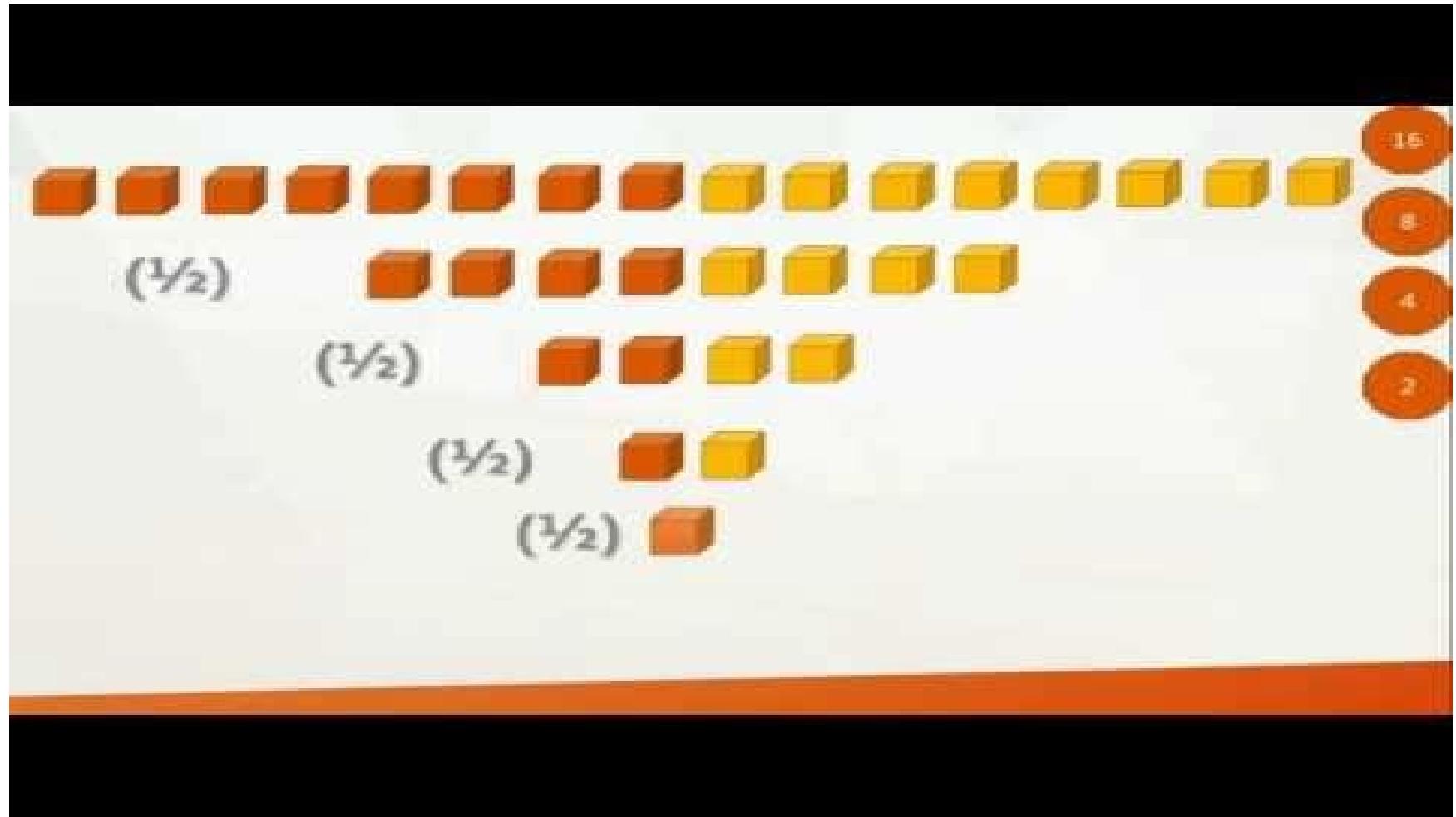
↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max



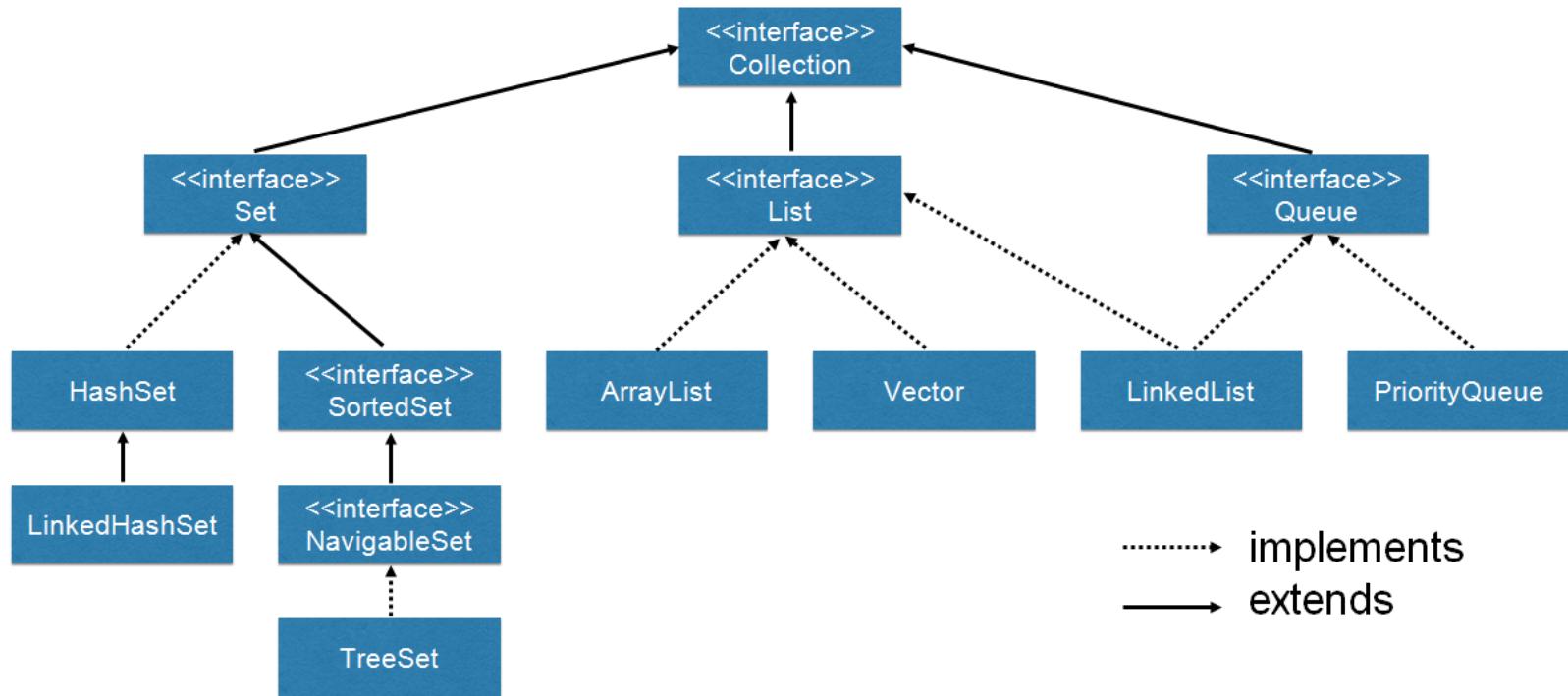
Source: <https://www.youtube.com/watch?v=b060MHQwUEE>

# How to get log growth?

- Starting at  $x=1$ , how many iterations of  $x^2$  before  $x \geq N$ ?
  - the *repeated doubling* principle
- Starting at  $x=N$ , how many iterations of  $x/2$  before  $x \leq 1$ ?
  - The *repeated halving* principle

# Collection interface

# Collection Interface



- a Collection **is** a data structure that holds items
  - very unspecific as to how the items are held
    - *i.e. the data structure*
- supports various operations:
  - add, remove, contains, ...
- examples:
  - ArrayList
  - PriorityQueue
  - LinkedList
  - TreeSet

# add

```
int[] d = new int[6];
```



size 0

# add

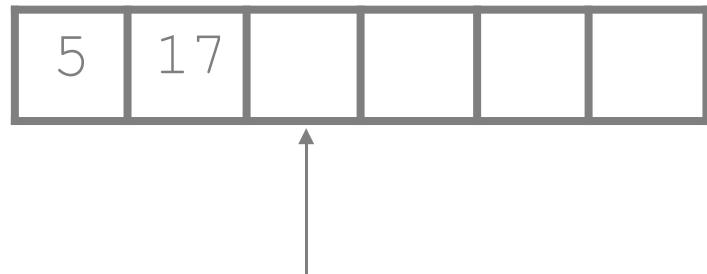
```
int[] d = new int[6];  
data.add(5);
```



size 1

# add

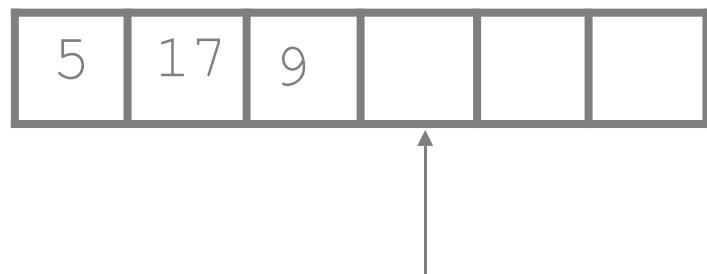
```
int[] d = new int[6];  
data.add(5);  
data.add(17);
```



size 2

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);
```

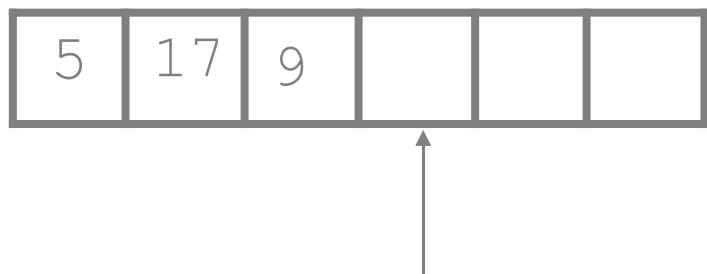


size 3

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);
```

Don't forget size++



size 3

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



size 6

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



```
data.add(22);
```

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



size 6

data.add(22);

Now what???

- We need to grow our array!
- Avoid allocating slightly larger arrays
  - You will most likely need to grow again soon
- Good rule of thumb is to double the size
  - *Con:* wastes up to 2x space
  - *Pro:* growth will be rare

# grow

data



```
tmp = new int[data.length*2];
```

tmp



# grow



```
tmp = new int[data.length*2];
```



copy all from data to tmp



# grow



```
tmp = new int[data.length*2];
```



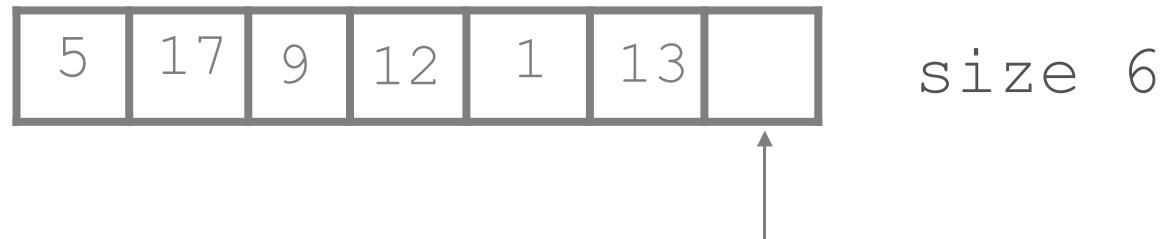
copy all from data to tmp



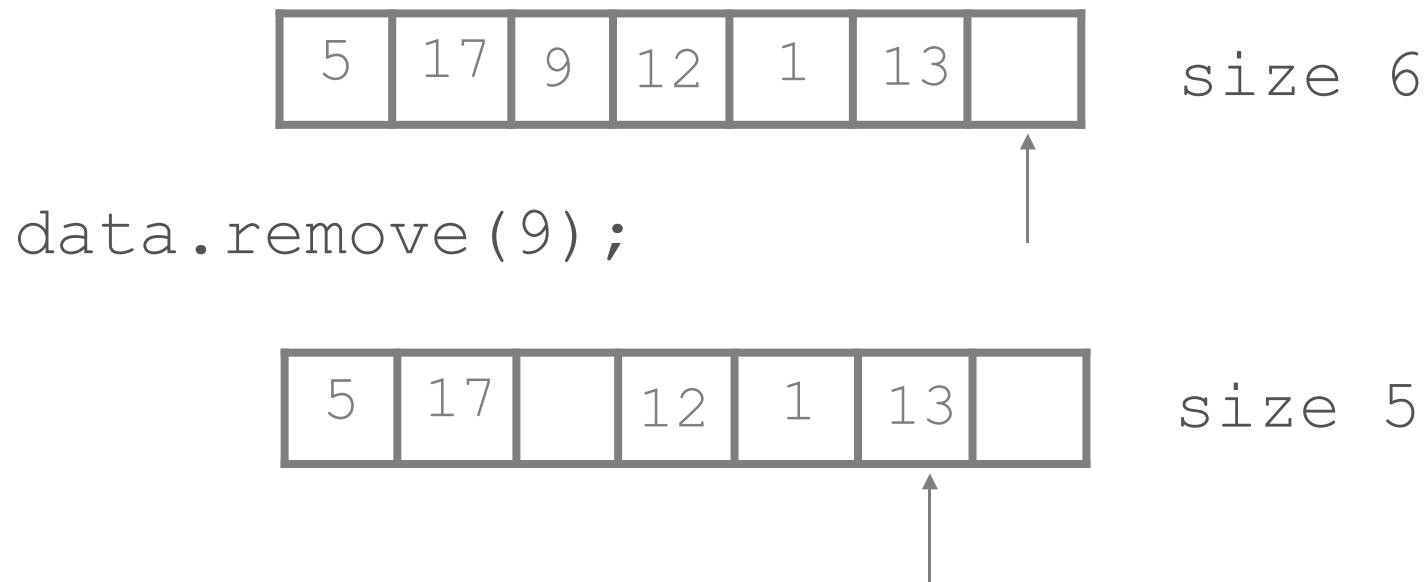
```
data = tmp;
```



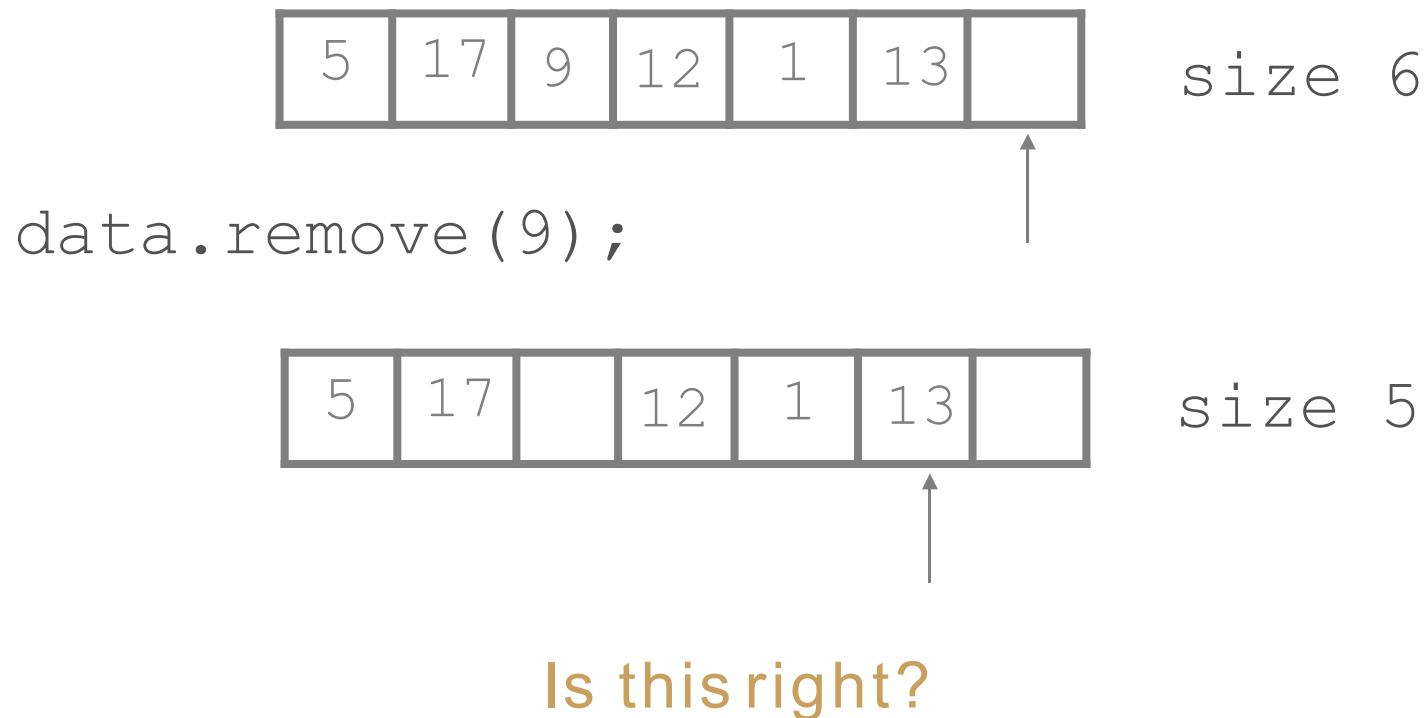
# remove



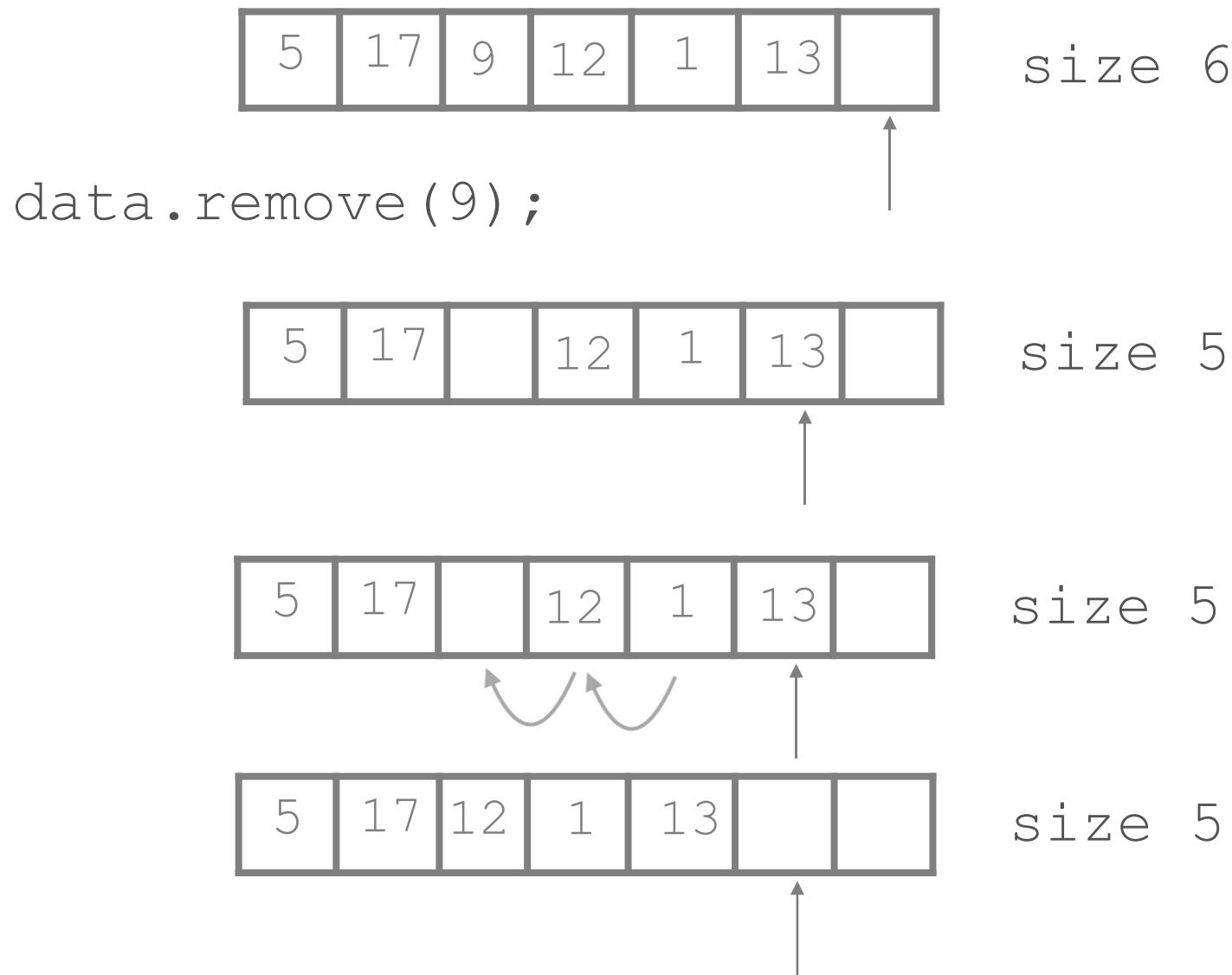
# remove



# remove

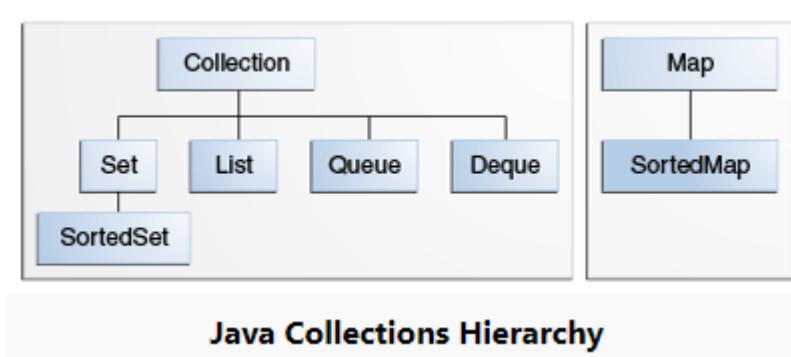


# remove



# Collection

- An object that groups multiple elements into a single unit
  - Store, retrieve, manipulate, and communicate stored data
  - Like an array except
    - Their size can change dynamically
    - Have more advanced behaviors



- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for (i=0; i<size; i++)
    data[i] ...
```

# iterators

- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for (i=0; i<size; i++)
    data[i] ...
```
- The **Iterator interface** provides generic retrieval of items from a data structure
  - Often times it is only sequential access

- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for(i=0; i<size; i++)
    data[i]...
```
- The **Iterator interface** provides generic retrieval of items from a data structure
  - Often times it is only sequential access
- The Collection interface **requires** an Iterator
  - For example, `ArrayList` has `iterator()` method that returns an Iterator

# iterator

- An Iterator is **specific** to a data structure, and **knows** how to traverse the structure

# iterator

- An Iterator is specific to a data structure, and knows how to traverse the structure
  - `hasNext`: determines if iteration is complete
  - `next`: gets the next item
  - `remove`: removes the last seen item

# iterator

- An Iterator is specific to a data structure, and knows how to traverse the structure
  - `hasNext`: determines if iteration is complete
  - `next`: gets the next item
  - `remove`: removes the last seen item
- Internally, keeps track of where the next item is (as well as other state)

# Next Time...

- **quiz** on Thursday

- From binary search, recursion and analysis of algorithm
- You have a lab tomorrow
- Get a new assignment tomorrow

# Analysis of Algorithm

## CSC220|Computer Programming 2

Correctness is only half the  
battle!!!

- Correctness is only half the battle
- Programs are expected to terminate in a reasonable amount of time

- Correctness is only half the battle
- Programs are expected to terminate in a reasonable amount of time
- Running time of a program is strongly correlated to the choice of algorithms used in problem solving

How much time and space does an algorithm require?

# Example

- Finding a word in a dictionary
- Algorithm 1
  - 1. Start on the **first** page, **first** entry
  - 2. If word not found, move to the next entry
  - 3. If very end of dictionary reached, word not found

# Example

- Finding a word in a dictionary
- Algorithm 1
  - 1. Start on the **first** page, **first** entry
  - 2. If word not found, move to the next entry
  - 3. If very end of dictionary reached, word not found

Does this work?  
Can we do any better?

# Example

- Finding a word in a dictionary
- Algorithm 2
  1. Guess which page the entry is on
  2. Did we go too far?
    - Go back some pages
  3. Did we not go far enough?
    - Go forward some pages
  4. Continue narrowing

what does this algorithm assume about the dictionary?

- Algorithm 1
  - Run time directly related to size of dictionary
    - Assume 100k words, each take .25sec → ~7hours!
- Algorithm 2
  - More like what humans do
  - A matter of seconds...

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

$1 \times 10^{12} \text{ *(minuscule amount of time) = large amount of time}$

# Who cares...

- Many different ways to solve a problem
  - One takes only 1ms longer than the other
  - What If I need to operate on LARGE number of items
    - Millions
    - Billions
    - ...

$1 \times 10^{12} * (\text{minuscule amount of time}) = \text{large amount of time}$

- That is why algorithm **complexity matters!**

# Complexity

- A measure of the computing resources that are used by a piece of code
  - Time
  - Memory
  - Disk space

# Complexity

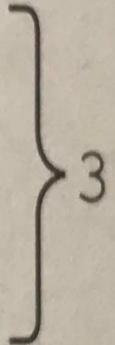
- Empirical analysis: timing how long it takes for a program to run
- Not very useful because different computers can run at different speeds
- Algorithm analysis is what we are after
  - The practice of applying techniques to mathematically approximate the performance of various computing algorithms

# N

- N is the problem size
  - *Sorting an array of N numbers*
  - *Searching for an item in a set of N items*
  - *Inserting an item into a set of N items*
- Amount of work done for these operations usually depends on N
  - Work required is a **function** of N
- Algorithms don't always require N steps for N items!

# Complexity

```
statement1.  
statement2.  
statement3.
```



3

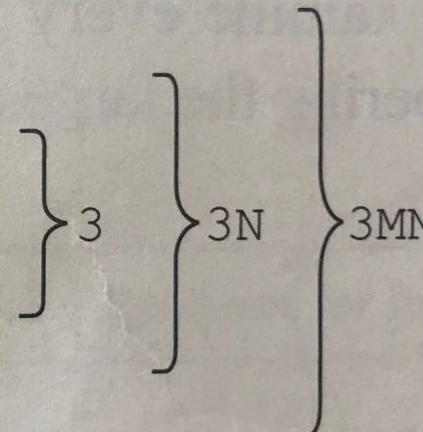
# Complexity

```
statement1.  
for (N times) { }  
    statement2. } N  
}  
  
for (M times) { }  
    statement3. } M  
}  
statement4.  
statement5.
```

The diagram illustrates the time complexity of a nested loop structure. It shows a sequence of statements: 'statement1.', followed by a 'for' loop that executes 'N' times, containing 'statement2.'. Below this is another 'for' loop that executes 'M' times, containing 'statement3.'. After these loops, there are two more statements: 'statement4.' and 'statement5.'. Braces on the right side group the code into three main sections: the single statement 'statement1.', the nested block of 'statement2.' within its loop, and the entire block of 'statement3.' within its loop. A large brace on the far right groups all five statements together. To the right of the first brace is the expression 'N'. To the right of the second brace is the expression 'M'. To the right of the third brace is the expression 'M + N + 3', representing the total number of operations.

# Complexity

```
for (M times) {  
    for (N times) {  
        statement1.  
        statement2.  
        statement3.  
    }  
}
```



# Complexity

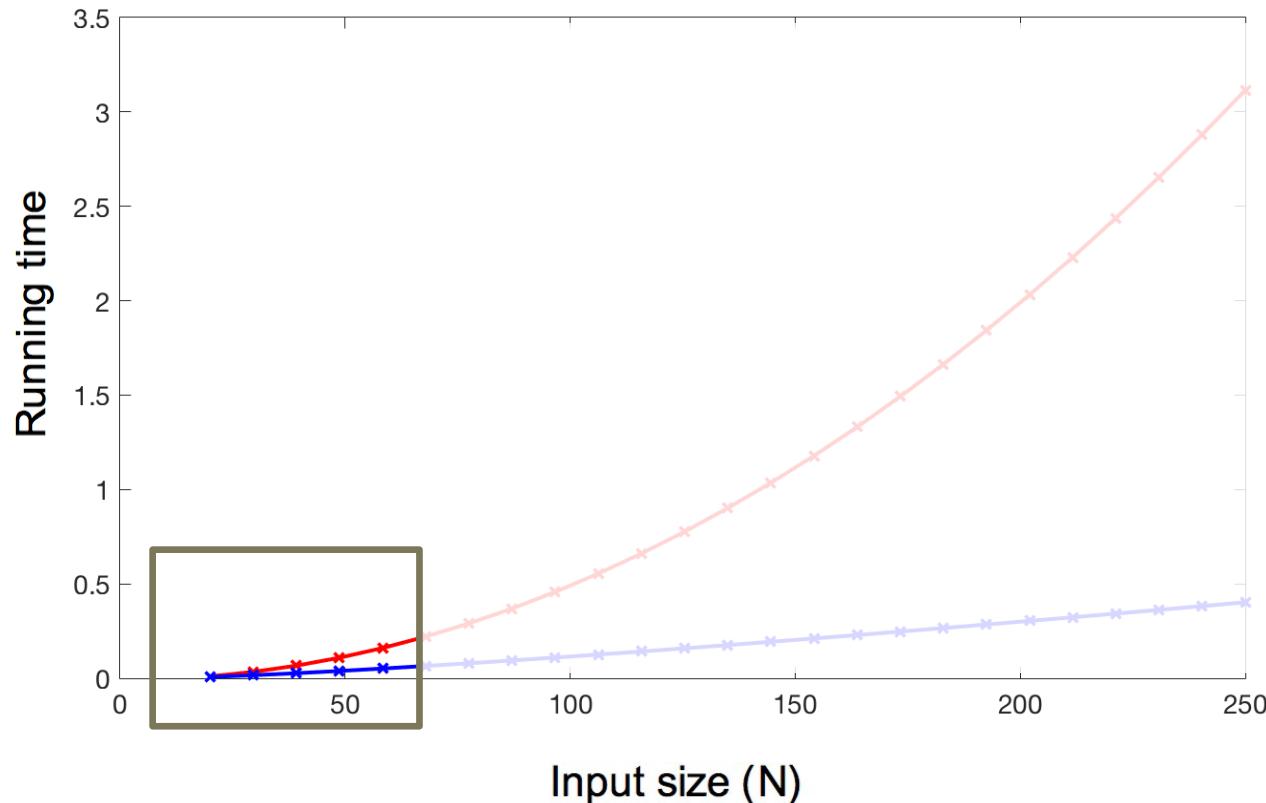
```
statement1.  
for (N times) {      }  
              statement2.  
    }                    } N  
  
for (M times) {      }  
              statement3.  
    }                    } M  
statement4.  
statement5.
```

The diagram illustrates the execution flow of the provided pseudocode. It features two nested loops: an outer loop (M times) containing an inner loop (N times). The code consists of five statements: statement1, statement2, statement3, statement4, and statement5. Statement1 is executed once. Statement2 is part of the inner loop and is executed  $N$  times. Statement3 is part of the outer loop and is executed  $M$  times. Statement4 and Statement5 are executed once each. Braces on the right side of the code group the loops and count the iterations. A large brace groups both loops and indicates the total complexity as  $M + N + 3$ .

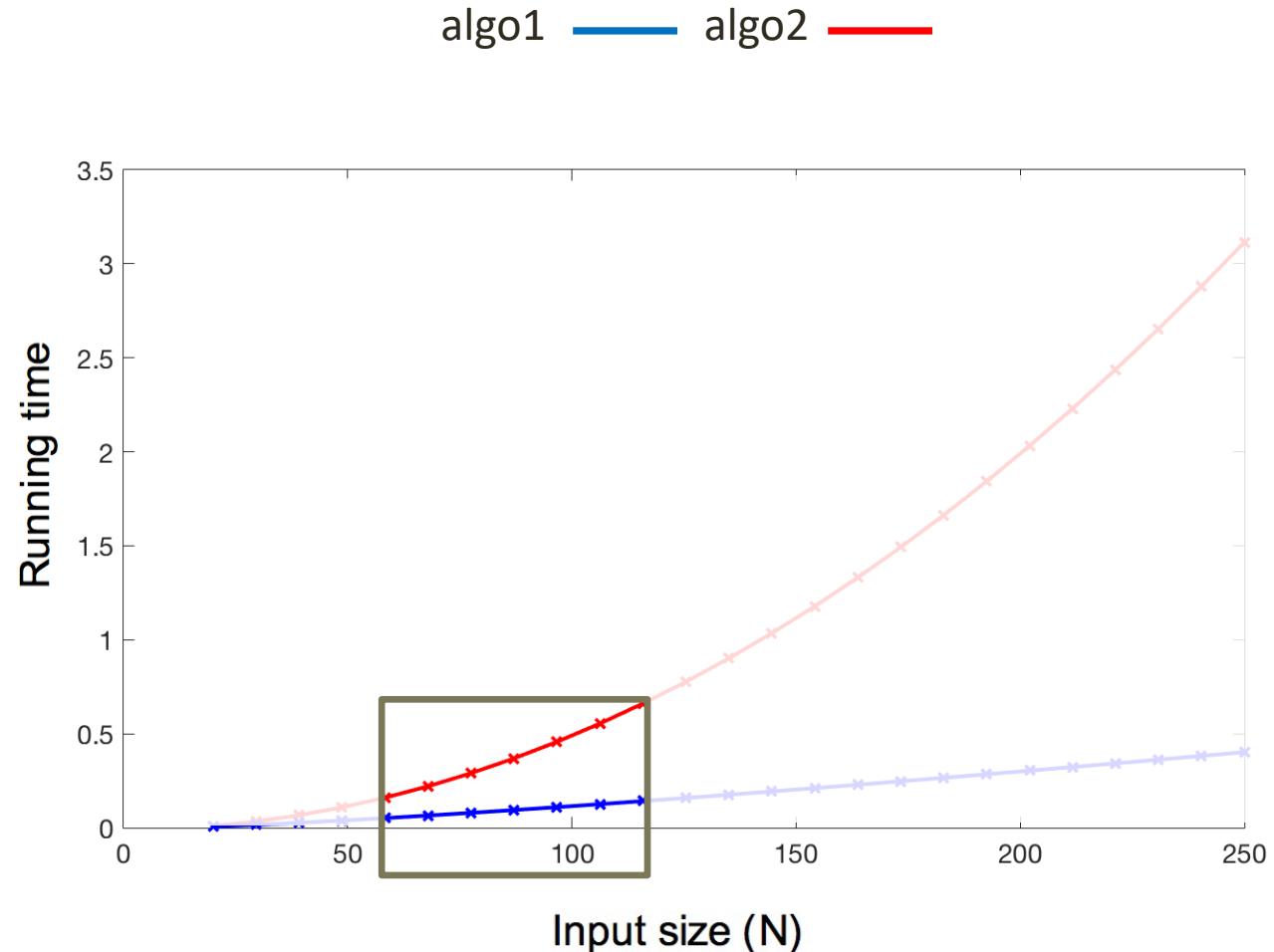
# Example

# small $N$

algo1 — algo2 —

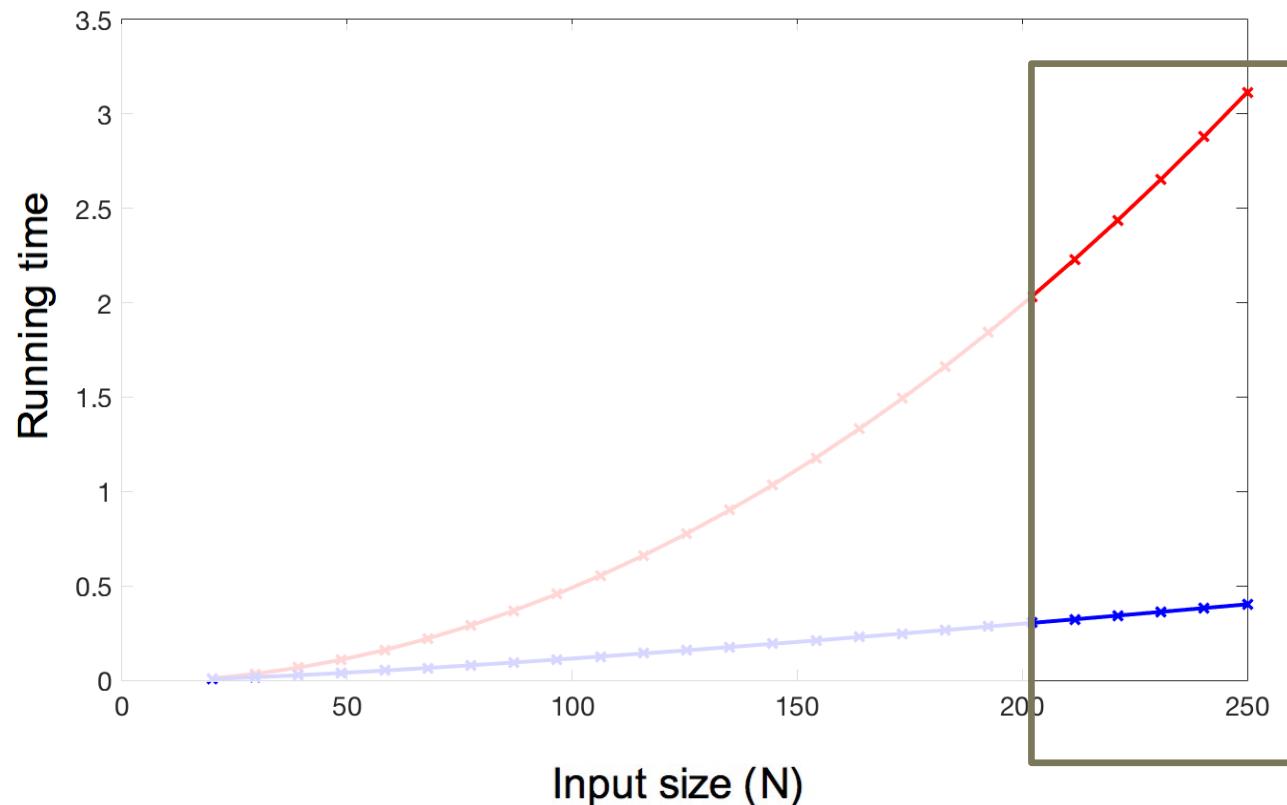


# medium N



# large N

algo1 — algo2 —



When **N** becomes large complexity  
matters!

# Growth Rate

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates

# Growth Rate

|            |              |
|------------|--------------|
| C          | constant     |
| $\log N$   | logarithmic  |
| $N$        | linear       |
| $N \log N$ | linearithmic |
| $N^2$      | quadratic    |
| $N^3$      | cubic        |

# Growth Rate

|            |              |
|------------|--------------|
| $C$        | constant     |
| $\log N$   | logarithmic  |
| $N$        | linear       |
| $N \log N$ | linearithmic |
| $N^2$      | quadratic    |
| $N^3$      | cubic        |

↓

Increasing growth rate

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates

- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates
- for sufficiently large  $N$ , a function's growth rate is determined by its dominate term

$N^2 + 1000N + 500$  —————> What is the dominate term?

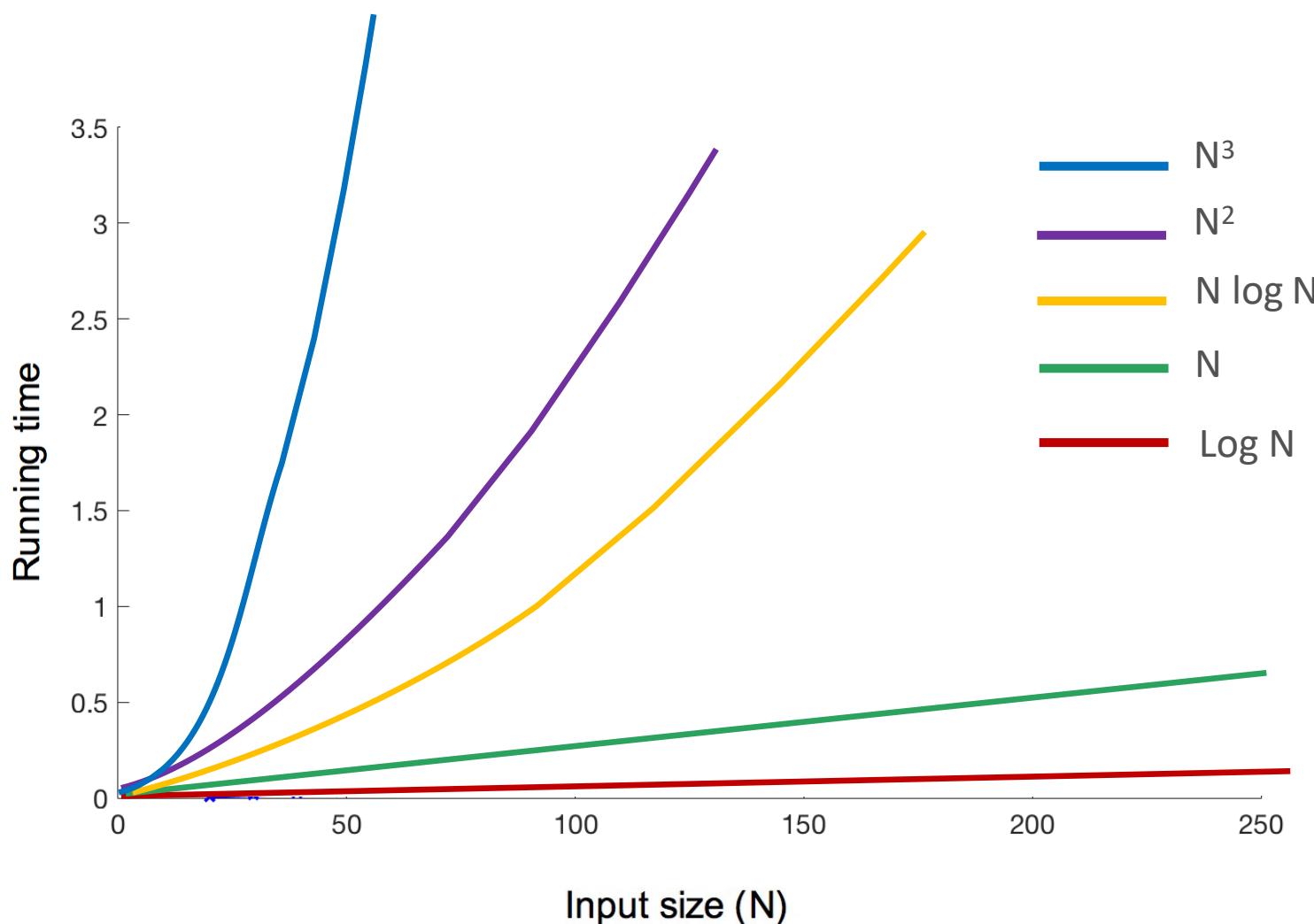
- knowing that  $F(N) < G(N)$  for a particular  $N$  is not very useful
  - instead, we measure the functions' growth rates
- for sufficiently large  $N$ , a function's growth rate is determined by its dominate term

|            |              |                                                                                                                |
|------------|--------------|----------------------------------------------------------------------------------------------------------------|
| $C$        | constant     | <br>Increasing growth rate |
| $\log N$   | logarithmic  |                                                                                                                |
| $N$        | linear       |                                                                                                                |
| $N \log N$ | linearithmic |                                                                                                                |
| $N^2$      | quadratic    |                                                                                                                |
| $N^3$      | cubic        |                                                                                                                |

# Big-O Notation

- **Big-O notation** ( $O$ ) is used to capture the dominate term in an algorithm
  - assuming large  $N$
- for example, the running time of a quadratic algorithm is  $N^2$  is specified  $O(N^2)$ 
  - read “order  $N$  squared”
- this notation allows us to establish a relative order among algorithms
  - $O(N \log N)$  is better than  $O(N^2)$

# Typical run-time complexities



- Constant -  $O(1)$ : have runtimes that don't depend on input size. Ex. Code to convert fahrenheit to celcius
- Logarithmic -  $O(\log N)$ : divide a problem space in half. Ex. Binary search
- Linear –  $O(N)$ : algorithms that are directly proportional to  $N$ . Ex. Counting, summing over a list of  $N$  items.

- Log Linear-  $O(N \log N)$ : have a combo of linear and log. Ex. Merge sort
- Quadratic-  $O(N^2)$ : runtimes proportional to the square of input size. Ex: double *for* loops
- Cubic–  $O(N^3)$ : runtimes proportional to the cube of input size. Ex. Triple for loops (3-d arrays)
- Exponential –  $O(2^N)$ : runtimes proportional to 2 raised to the power of N. EX: yeah, don't do these unless it's a small N.

# Example

- Finding the maximum item in an array
  - Initialize max to the first element
  - Scan through each item in the array
    - If the item is greater than max, update max

What is the big-o complexity of this algorithm?

- 1.c
- 2.log N
- 3.N
- 4.N log N
- 5.N<sup>2</sup>
- 6.N<sup>3</sup>

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;
for(int i=0; i<array.length-1; i++) {
    int num1 = array[i];
    for(int j=i+1; j<array.length; j++) {
        int num2 = array[j];
        if (Math.abs(num1-num2) < diff)
            diff = Math.abs(num1-num2);
    }
}
return diff;
```

What is the big-o complexity of this algorithm?

# Example

- Finding the smallest difference

```
int diff = Integer.MAX_VALUE;  
for(int i=0; i<array.length-1; i++) {  
    int num1 = array[i];  
    for(int j=i+1; j<array.length; j++) {  
        int num2 = array[j];  
        if (Math.abs(num1-num2) < diff)  
            diff = Math.abs(num1-num2);  
    }  
}  
  
return diff;
```

What is the big-o complexity of this algorithm?

c  
 $\log N$   
 $N$   
 $N \log N$   
 $N^2$   
 $N^3$

**O(n<sup>2</sup>)**

```
1 // C++ program to find minimum difference between
2 // any pair in an unsorted array
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 // Returns minimum difference between any pair
7 int findMinDiff(int arr[], int n)
8 {
9     // Sort array in non-decreasing order
10    sort(arr, arr+n);
11
12    // Initialize difference as infinite
13    int diff = INT_MAX;
14
15    // Find the min diff by comparing adjacent
16    // pairs in sorted array
17    for (int i=0; i<n-1; i++)
18        if (arr[i+1] - arr[i] < diff)
19            diff = arr[i+1] - arr[i];
20
21    // Return min diff
22    return diff;
23}
24
25 // Driver code
26 int main()
27 {
28     int arr[] = {1, 5, 3, 19, 18, 25};
29     int n = sizeof(arr)/sizeof(arr[0]);
30     cout << "Minimum difference is " << findMinDiff(arr, n);
31     return 0;
32}
```

## Method 2 (Efficient: O(n Log n))

The idea is to use sorting. Below are steps.

- 1) Sort array in ascending order. This step takes  $O(n \log n)$  time.
- 2) Initialize difference as infinite. This step takes  $O(1)$  time.
- 3) Compare all adjacent pairs in sorted array and keep track of minimum difference. This step takes  $O(n)$  time.

# Analyze the running time

```
for(int i=0; i<n; i+=2)  
    sum++;
```

1.c

```
for(int i=0; i<n; i++)  
    for(int j=0; j<n*n; j++)  
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

# Analyze the running time

```
for(int i=0; i<n; i+=2)
    sum++;
```

1.c

```
for(int i=0; i<n; i++)
    for(int j=0; j<n*n; j++)
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

# Analyze the running time

```
for(int i=0; i<n; i+=2)  
    sum++;
```

1.c

```
for(int i=0; i<n; i++)  
    for(int j=0; j<n*n; j++)  
        sum++;
```

2.log N

3.N

4.N log N

5.N<sup>2</sup>

6.N<sup>3</sup>

What is the complexity of binary search?

# Binary search

```
public static int binarySearch(int[] numbers, int target){  
    int min = 0;  
    int max = numbers.length-1;  
    while (min <= max){  
        int mid = (max + min)/2;  
        if (numbers[mid] == target){  
            return mid; // found it!  
        }else if (numbers[mid] < target){  
            min = mid + 1; // too small  
        }else{ //numbers[mid] > target  
            max = mid - 1; // too large  
        }  
    }  
    return -1;  
}
```

# Binary search

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

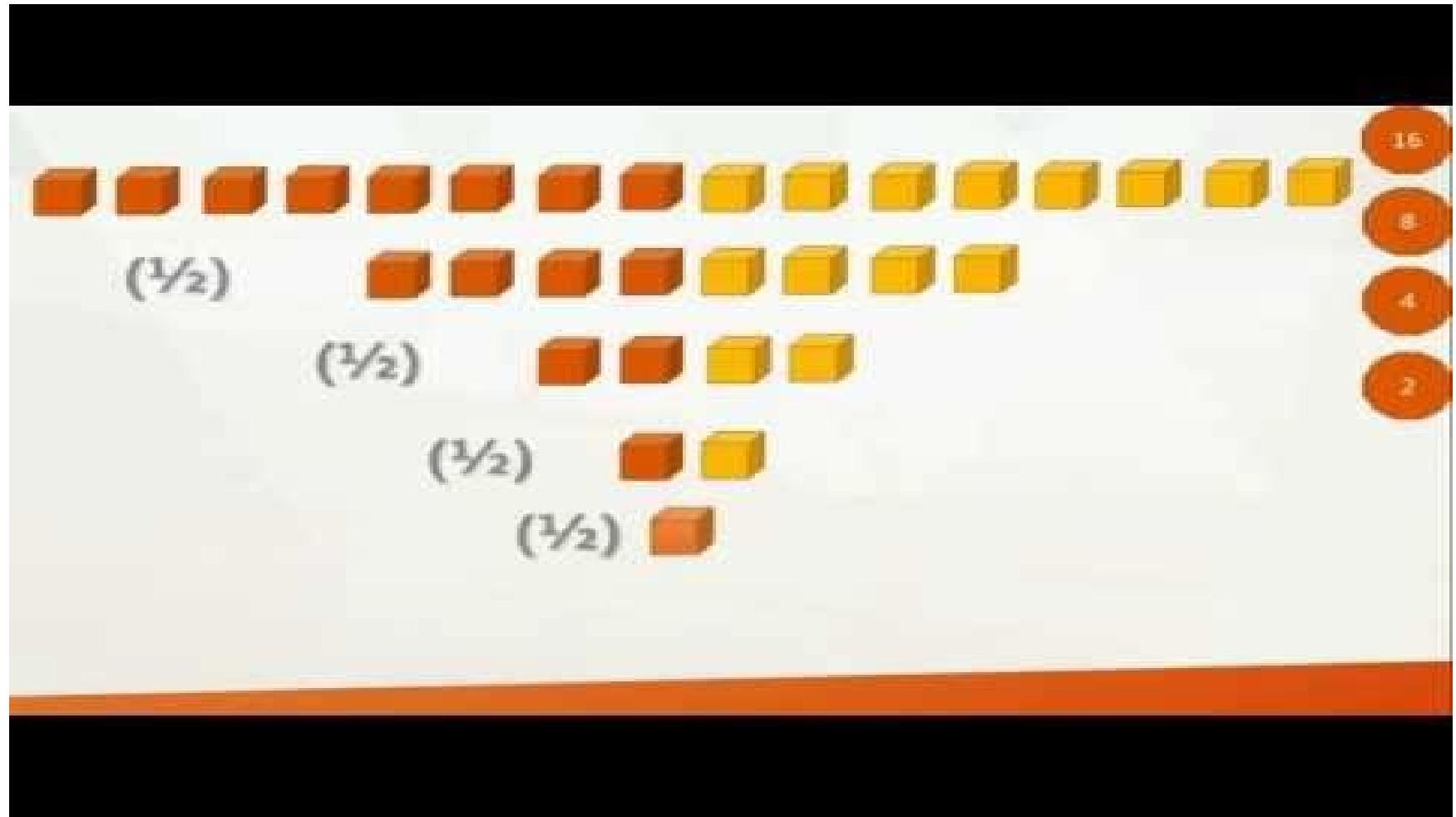
↑  
min

↑  
mid

↑  
max

|   |    |    |    |    |    |    |    |    |    |     |
|---|----|----|----|----|----|----|----|----|----|-----|
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|----|----|----|----|----|----|----|----|----|-----|

↑  
min  
mid  
max



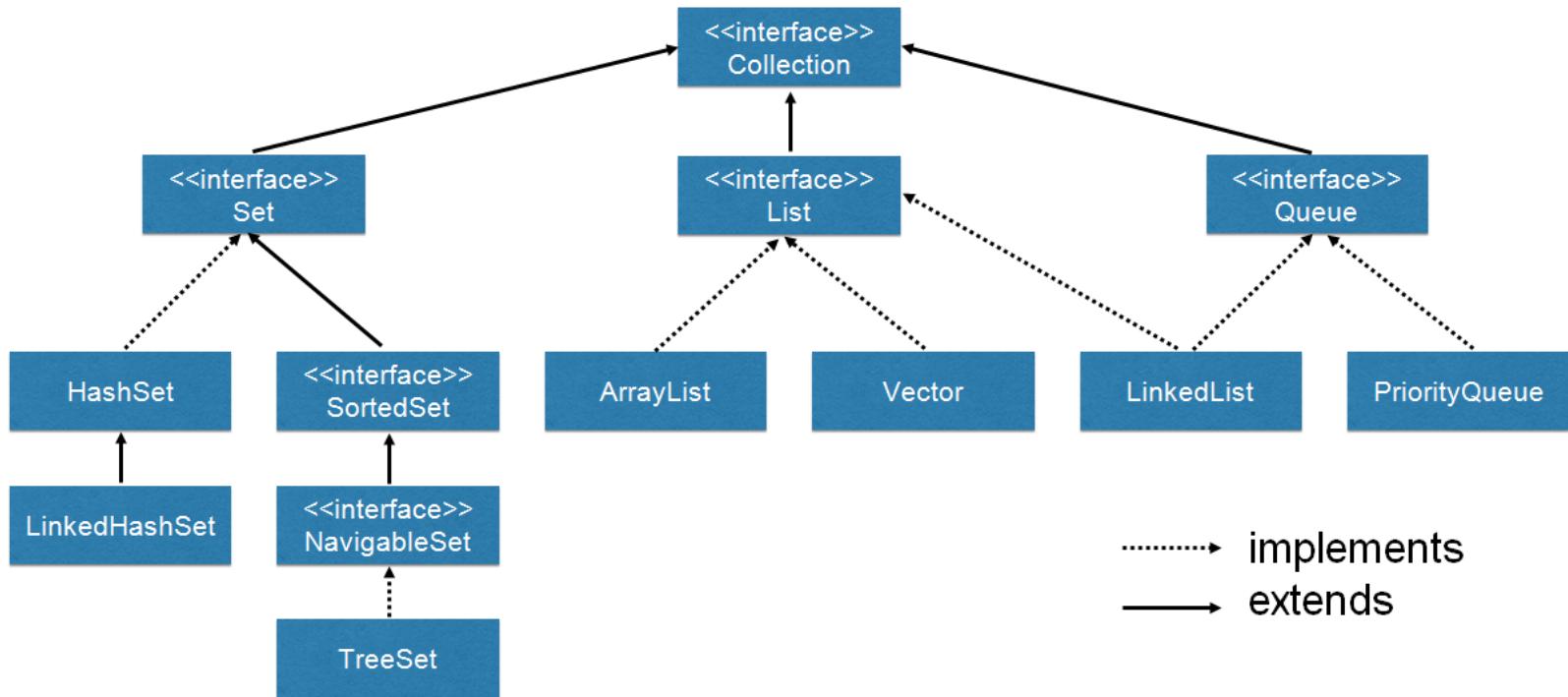
Source: <https://www.youtube.com/watch?v=b060MHQwUEE>

# How to get log growth?

- Starting at  $x=1$ , how many iterations of  $x^2$  before  $x \geq N$ ?
  - the *repeated doubling* principle
- Starting at  $x=N$ , how many iterations of  $x/2$  before  $x \leq 1$ ?
  - The *repeated halving* principle

# Collection interface

# Collection Interface



- a Collection **is** a data structure that holds items
  - very unspecific as to how the items are held
    - *i.e. the data structure*
- supports various operations:
  - add, remove, contains, ...
- examples:
  - ArrayList
  - PriorityQueue
  - LinkedList
  - TreeSet

# add

```
int[] d = new int[6];
```



size 0

# add

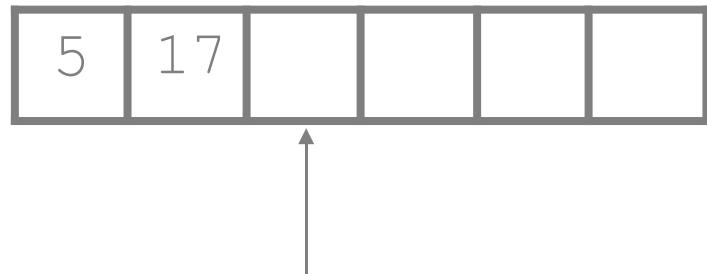
```
int[] d = new int[6];  
data.add(5);
```



size 1

# add

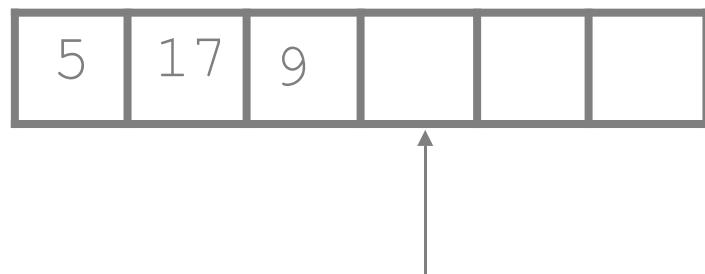
```
int[] d = new int[6];  
data.add(5);  
data.add(17);
```



size 2

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);
```

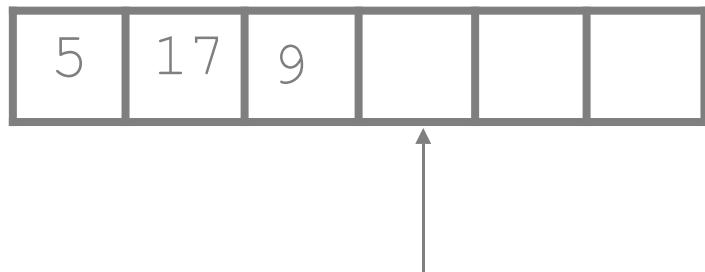


size 3

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);
```

Don't forget size++



size 3

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



size 6



# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



size 6

```
data.add(22);
```

# add

```
int[] d = new int[6];  
data.add(5);  
data.add(17);  
data.add(9);  
data.add(12);  
data.add(1);  
data.add(13);
```



size 6

data.add(22);

Now what???

- We need to grow our array!
- Avoid allocating slightly larger arrays
  - You will most likely need to grow again soon
- Good rule of thumb is to double the size
  - *Con:* wastes up to 2x space
  - *Pro:* growth will be rare

# grow

data



```
tmp = new int[data.length*2];
```

tmp



# grow



```
tmp = new int[data.length*2];
```



copy all from data to tmp



# grow



```
tmp = new int[data.length*2];
```



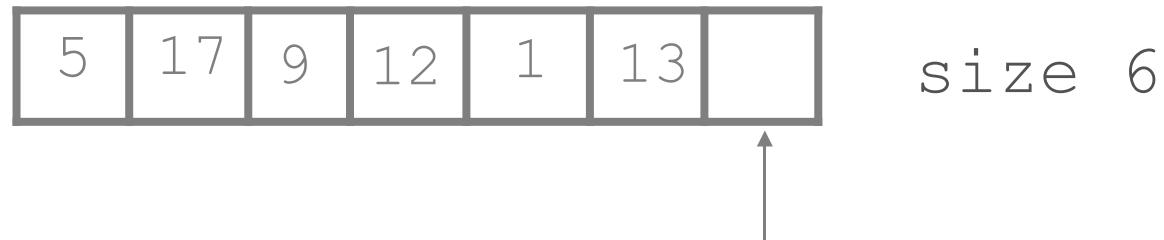
copy all from data to tmp



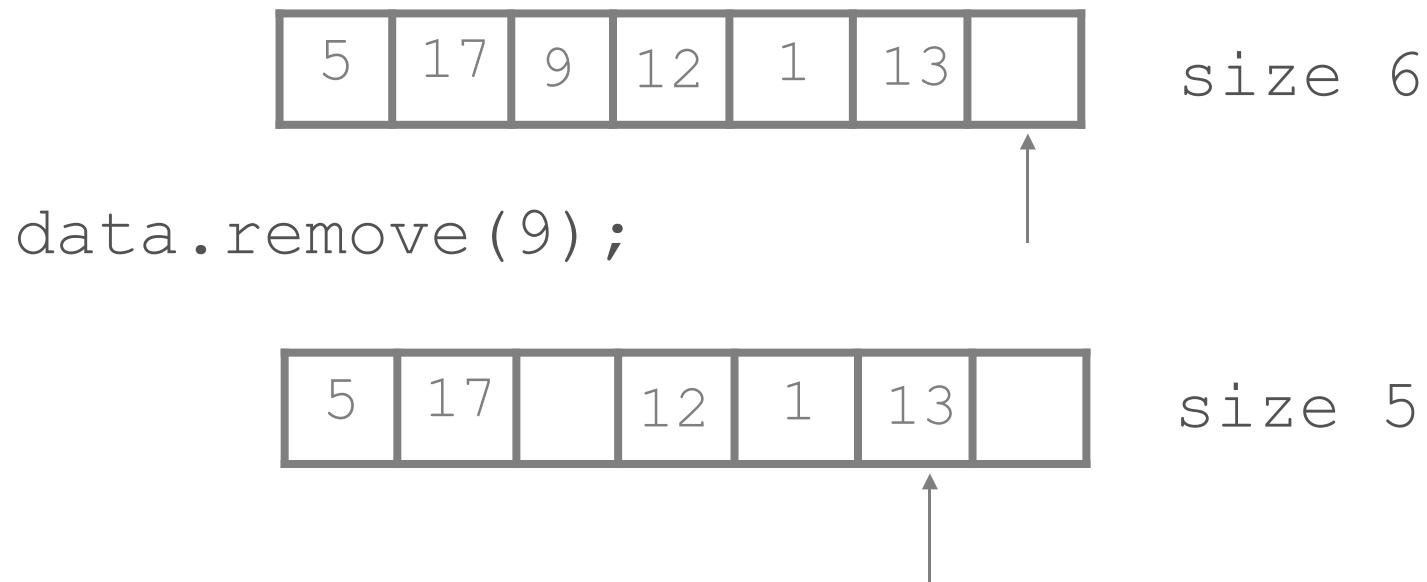
```
data = tmp;
```



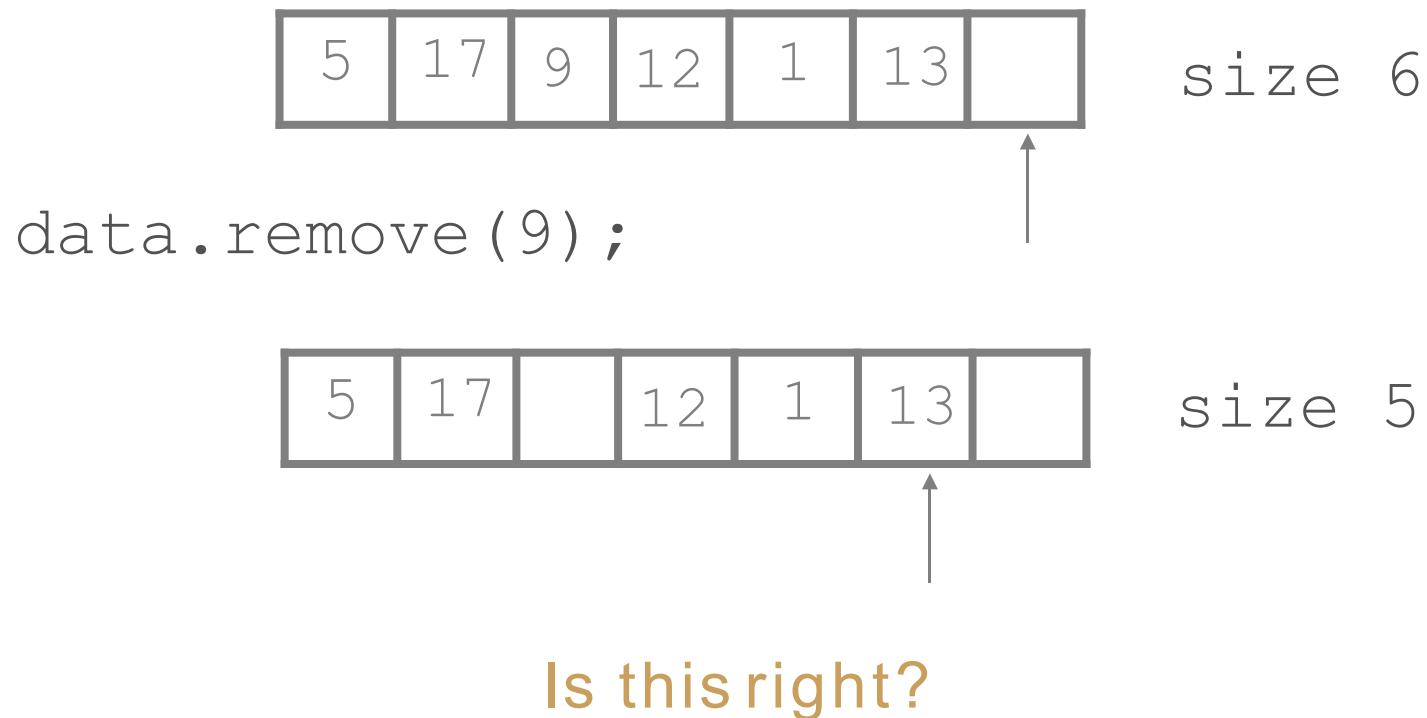
# remove



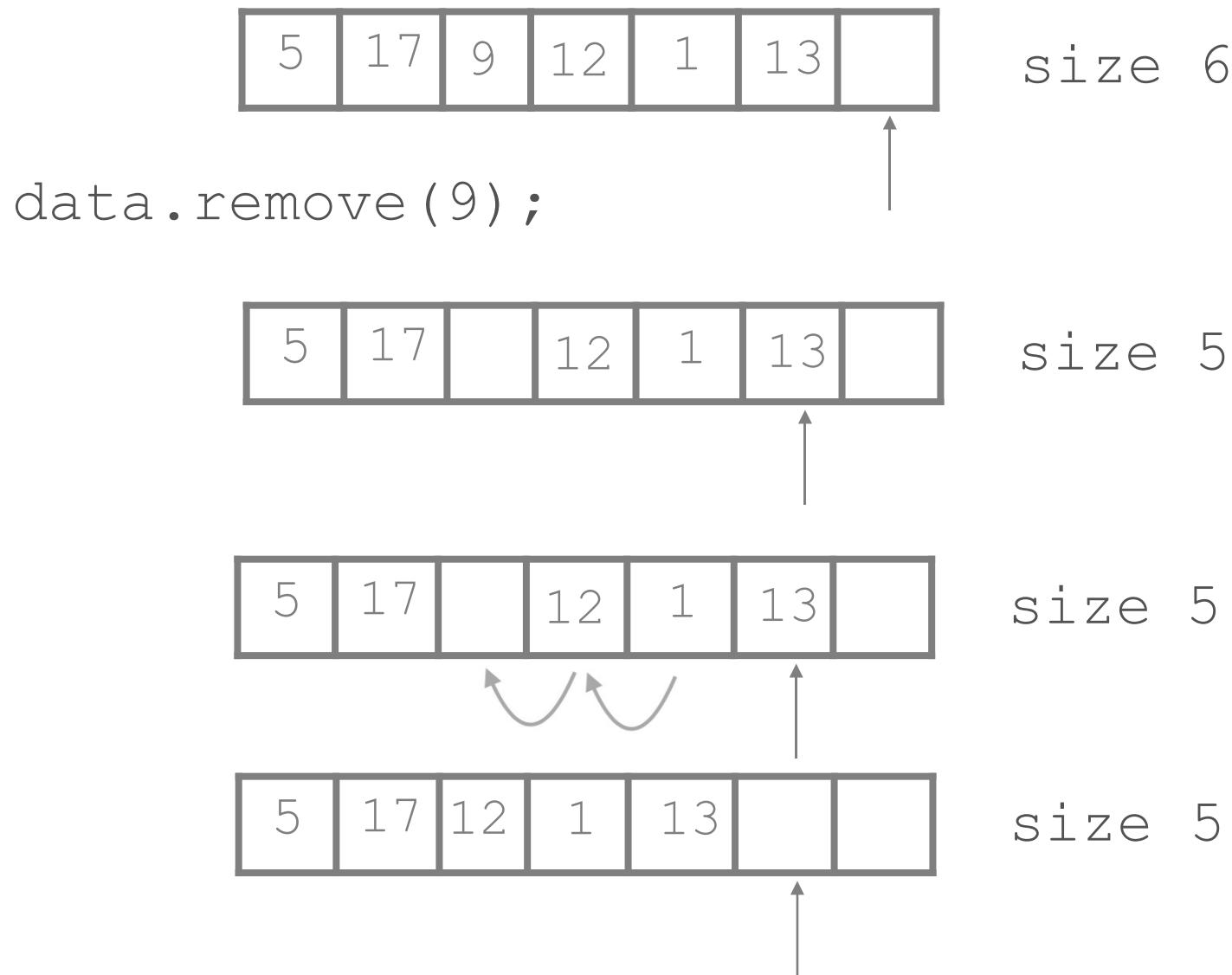
# remove



# remove

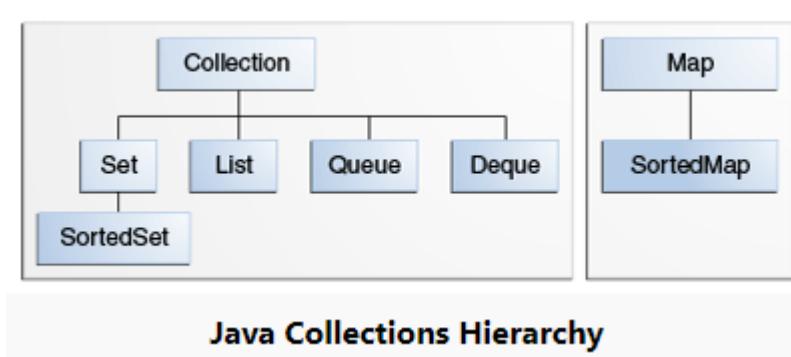


# remove



# Collection

- An object that groups multiple elements into a single unit
  - Store, retrieve, manipulate, and communicate stored data
  - Like an array except
    - Their size can change dynamically
    - Have more advanced behaviors



- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for (i=0; i<size; i++)
    data[i] ...
```

# iterators

- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for (i=0; i<size; i++)
    data[i] ...
```
- The **Iterator interface** provides generic retrieval of items from a data structure
  - Often times it is only sequential access

- Not all data structures are guaranteed to use an array
  - Thus, we can't just do:

```
for(i=0; i<size; i++)
    data[i]...
```
- The **Iterator interface** provides generic retrieval of items from a data structure
  - Often times it is only sequential access
- The Collection interface **requires** an Iterator
  - For example, `ArrayList` has `iterator()` method that returns an Iterator

# iterator

- An Iterator is **specific** to a data structure, and **knows** how to traverse the structure

# iterator

- An Iterator is specific to a data structure, and knows how to traverse the structure
  - `hasNext`: determines if iteration is complete
  - `next`: gets the next item
  - `remove`: removes the last seen item

# iterator

- An Iterator is specific to a data structure, and knows how to traverse the structure
  - `hasNext`: determines if iteration is complete
  - `next`: gets the next item
  - `remove`: removes the last seen item
- Internally, keeps track of where the next item is (as well as other state)

# Next Time...

- **quiz** on Thursday

- From binary search, recursion and analysis of algorithm
- You have a lab tomorrow
- Get a new assignment tomorrow

# Analysis of Algorithm and Sorting

## CSC220 | Computer Programming 2

Last Time...

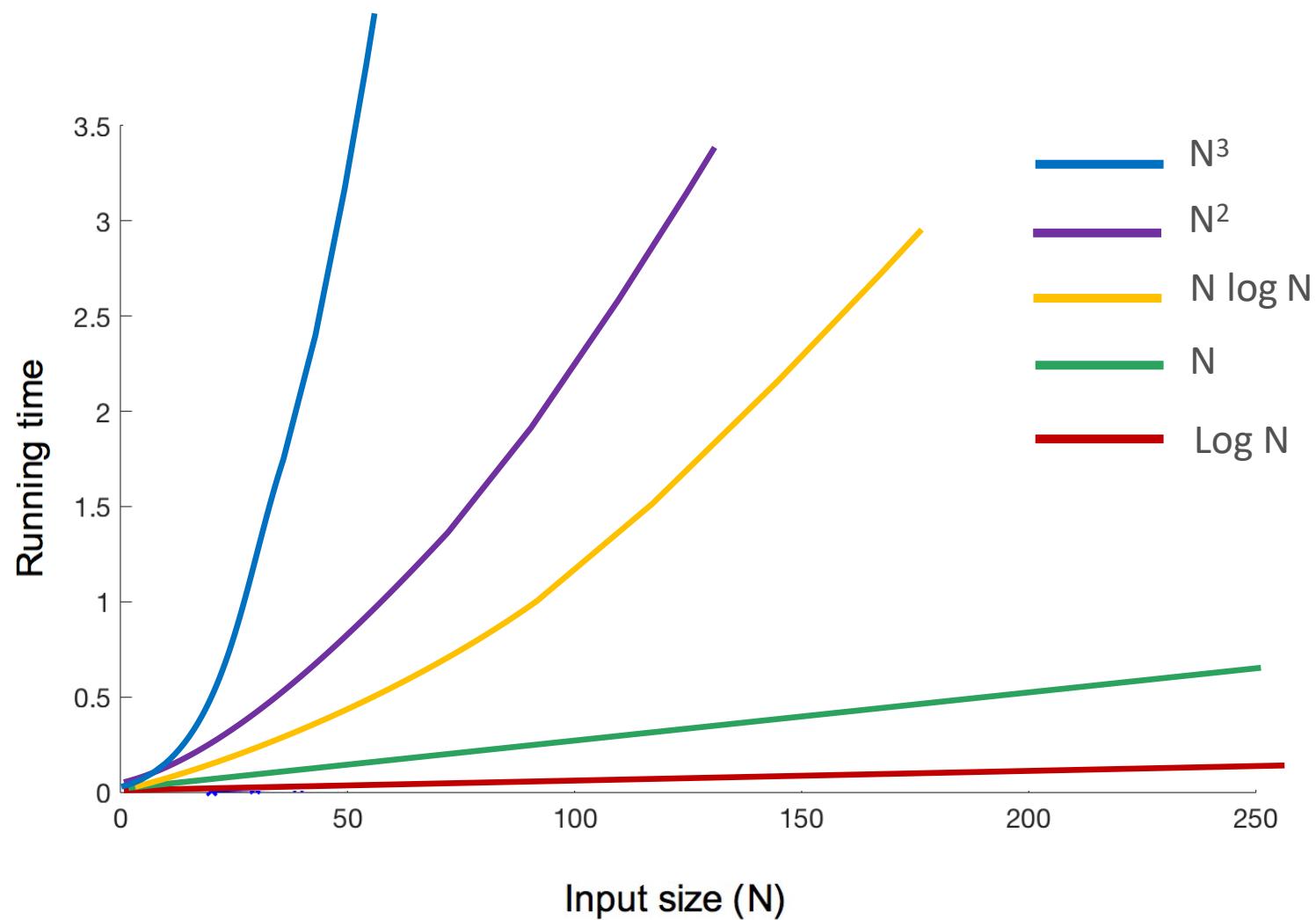
Correctness is only half the battle!!!

# Complexity

- A measure of the computing resources that are used by a piece of code
  - Time
  - Memory
  - Disk space

- **Big-O notation** ( $O$ ) is used to capture the dominate term in an algorithm
  - assuming large  $N$
- for example, the running time of a quadratic algorithm is  $N^2$  is specified  $O(N^2)$ 
  - read “order N squared”
- this notation allows us to establish a relative order among algorithms
  - $O(N \log N)$  is better than  $O(N^2)$

# Typical run-time complexities



# iterator

- Moving this to when we go over Linked Lists in a few weeks.
- Table it for now.

Today...

# Sorting

- Sorting is a fundamental application in computing
  - One of the most intensively studied and important operations
- Most data is useless unless it is in some kind of order

- Sorting is a fundamental application in computing
  - One of the most intensively studied and important operations
- Most data is useless unless it is in some kind of order
- For any given problem, or specific goal isn't necessarily sorting... but we often need to sort to efficiently solve problems
  - Computer graphics
  - Look-up tables
  - Games

- Sorting algorithms that are easy to understand  
(and implement) run in **quadratic time**

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!
- For very specific types of data we can actually do better

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!
- For very specific types of data we can actually do better
  - But we won't study these algorithms extensively

without thinking too hard, how can we sort  
any array of items?

# Selection Sort

The simplest sorting algorithm

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Selection sort

```
public static void selectionSort(int[] arr){  
    for(int i=0; i < arr.length-1; i++){  
        int min = i;  
        for(int j=i+1; j < arr.length; j++){  
            if (arr[j] < arr[min])  
                min = j;  
        }  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ←———— Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ←———— Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

Look for items less than those in sorted part of array

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ← Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp; ] Swap the items  
    }  
}
```

Look for items less than those in sorted part of array

What is the complexity?

# Selection sort

```
public static void selectionSort(int[] arr){  
    for(int i=0; i < arr.length-1; i++){  
        int min = i;  
        for(int j=i+1; j < arr.length; j++){  
            if (arr[j] < arr[min])  
                min = j;  
        }  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i;  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```



Source: [https://www.youtube.com/watch?v=f8hXR\\_Hvybo](https://www.youtube.com/watch?v=f8hXR_Hvybo)

# Class Activity

# Insertion Sort

Good for small  $\mathbf{N}$

# Insertion sort

Good for small N

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

# Insertion sort

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←————
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index; Insert item
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What the complexity of insertion sort  
(in the worst case)?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

Best case!



Source: <https://www.youtube.com/watch?v=DFG-XuyPYUQ>

# Class Activity

Next Time...

- Next week
  - Continue on sorting
    - Learning about other popular sort algorithms
  - Start on your assignment early

# Analysis of Algorithm and Sorting

## CSC220 | Computer Programming 2

Last Time...

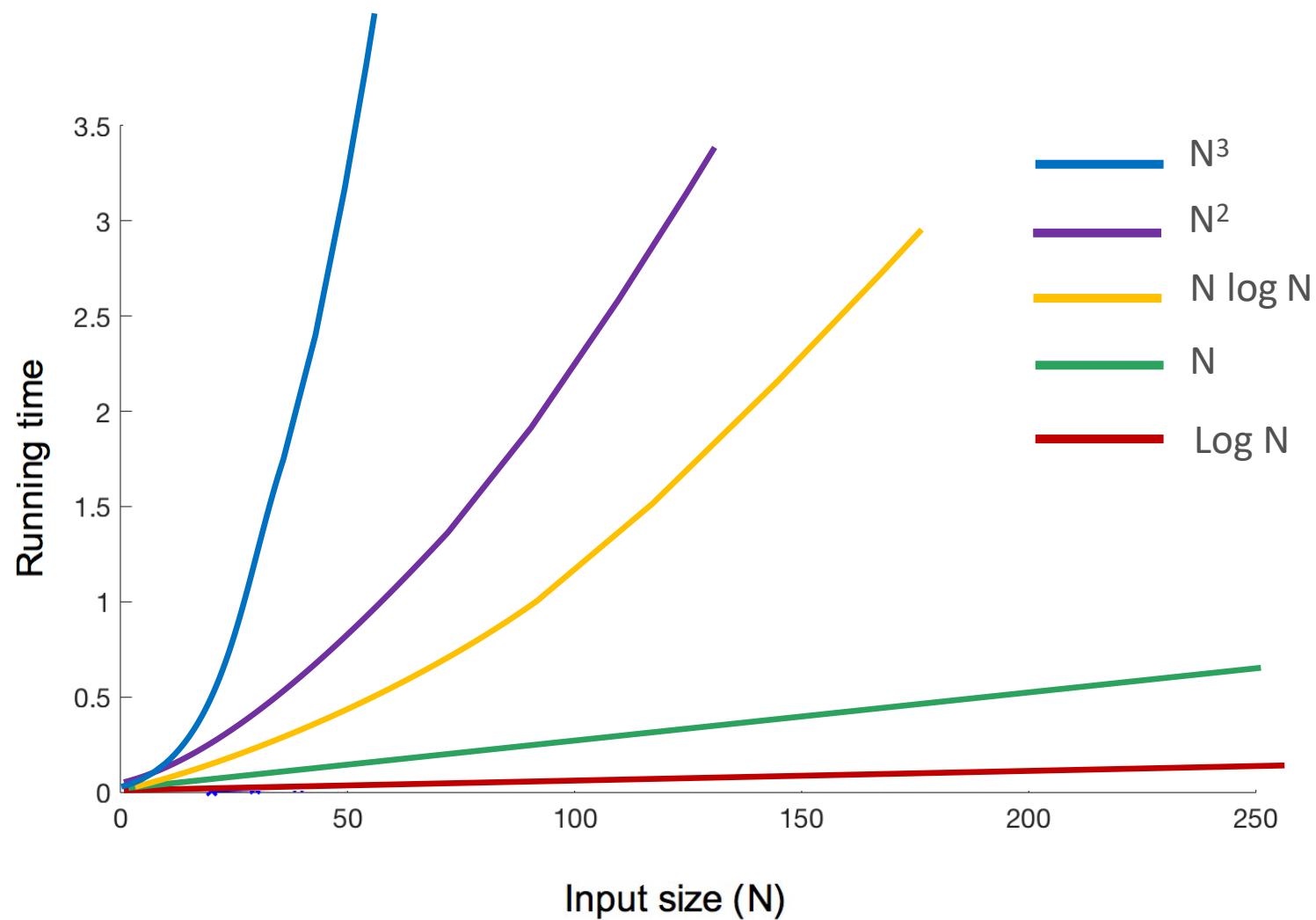
Correctness is only half the battle!!!

# Complexity

- A measure of the computing resources that are used by a piece of code
  - Time
  - Memory
  - Disk space

- **Big-O notation** ( $O$ ) is used to capture the dominate term in an algorithm
  - assuming large  $N$
- for example, the running time of a quadratic algorithm is  $N^2$  is specified  $O(N^2)$ 
  - read “order N squared”
- this notation allows us to establish a relative order among algorithms
  - $O(N \log N)$  is better than  $O(N^2)$

# Typical run-time complexities



# iterator

- Moving this to when we go over Linked Lists in a few weeks.
- Table it for now.

Today...

# Sorting

- Sorting is a fundamental application in computing
  - One of the most intensively studied and important operations
- Most data is useless unless it is in some kind of order

- Sorting is a fundamental application in computing
  - One of the most intensively studied and important operations
- Most data is useless unless it is in some kind of order
- For any given problem, or specific goal isn't necessarily sorting... but we often need to sort to efficiently solve problems
  - Computer graphics
  - Look-up tables
  - Games

- Sorting algorithms that are easy to understand  
(and implement) run in **quadratic time**

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!
- For very specific types of data we can actually do better

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  **$O(N \log N)$** 
  - Implementation details are critical to attaining this bound!
- For very specific types of data we can actually do better
  - But we won't study these algorithms extensively

without thinking too hard, how can we sort  
any array of items?

# Selection Sort

The simplest sorting algorithm

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Selection sort

```
public static void selectionSort(int[] arr){  
    for(int i=0; i < arr.length-1; i++){  
        int min = i;  
        for(int j=i+1; j < arr.length; j++){  
            if (arr[j] < arr[min])  
                min = j;  
        }  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ←———— Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ←———— Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

Look for items less than those in sorted part of array

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ← Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp; ] Swap the items  
    }  
}
```

Look for items less than those in sorted part of array

What is the complexity?

# Selection sort

```
public static void selectionSort(int[] arr){  
    for(int i=0; i < arr.length-1; i++){  
        int min = i;  
        for(int j=i+1; j < arr.length; j++){  
            if (arr[j] < arr[min])  
                min = j;  
        }  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i;  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```



Source: [https://www.youtube.com/watch?v=f8hXR\\_Hvybo](https://www.youtube.com/watch?v=f8hXR_Hvybo)

# Class Activity

# Insertion Sort

Good for small  $\mathbf{N}$

# Insertion sort

Good for small N

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

# Insertion sort

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←————
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index; Insert item
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What the complexity of insertion sort  
(in the worst case)?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

Best case!



Source: <https://www.youtube.com/watch?v=DFG-XuyPYUQ>

# Class Activity

Next Time...

- Next week
  - Continue on sorting
    - Learning about other popular sort algorithms
  - Start on your assignment early

# Sorting

## CSC220 | Computer Programming 2

Last Time...

- Let's do some more examples of Big-O notation

- What O are these?

```
1 | int n = 1000;  
2 | System.out.println("Hey - your input is: " + n);
```

- What O are these?

```
1 int n = 1000;
2 System.out.println("Hey - your input is: " + n);
3 System.out.println("Hmm.. I'm doing more stuff with: " + n);
4 System.out.println("And more: " + n);
```

- What O are these?

```
1 int n = 1000;  
2 System.out.println("Hey - your input is: " + n);  
3 System.out.println("Hmm.. I'm doing more stuff with: " + n);  
4 System.out.println("And more: " + n);
```

$O(1)$  - it *doesn't depend on the size of the input, n*

- What O are these?

```
1 | for (int i = 0; i < n; i++) {  
2 |     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

VS.

```
1 | for (int i = 1; i < n; i = i * 2){  
2 |     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

- What O are these?

$O(n)$

```
1 | for (int i = 0; i < n; i++) {  
2 |     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

VS.

$O(\log(n))$

```
1 | for (int i = 1; i < n; i = i * 2){  
2 |     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

If  $n$  is 8, the output will be the following:

```
1 | Hey - I'm busy looking at: 1  
2 | Hey - I'm busy looking at: 2  
3 | Hey - I'm busy looking at: 4
```

Our simple algorithm ran  $\log(8) = 3$  times.

- What O are these?

```
1  for (int i = 0; i < n; i++) {  
2      System.out.println("Hey - I'm busy looking at: " + i);  
3      System.out.println("Hmm.. Let's have another look at: " + i);  
4      System.out.println("And another: " + i);  
5  }
```

- What O are these?

```
1  for (int i = 0; i < n; i++) {  
2      System.out.println("Hey - I'm busy looking at: " + i);  
3      System.out.println("Hmm.. Let's have another look at: " + i);  
4      System.out.println("And another: " + i);  
5  }
```

$O(n)$

- What O are these?

```
1 | for (int i = 1; i <= n; i++) {  
2 |     for(int j = 1; j < 8; j = j * 2) {  
3 |         System.out.println("Hey - I'm busy looking at: " + i + " and " + j);  
4 |     }  
5 | }
```

VS.

```
1 | for (int i = 1; i <= Math.pow(2, n); i++) {  
2 |     System.out.println("Hey - I'm busy looking at: " + i);  
3 | }
```

- What O are these?

$O(n)$

```
1 for (int i = 1; i <= n; i++) {  
2     for(int j = 1; j < 8; j = j * 2) {  
3         System.out.println("Hey - I'm busy looking at: " + i + " and " + j);  
4     }  
5 }
```

VS.

$O(2^n)$

```
1 for (int i = 1; i <= Math.pow(2, n); i++) {  
2     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

- What O are these?

```
1 for (int i = 1; i <= n; i++) {  
2     for(int j = 1; j < n j = j * 2) {  
3         System.out.println("Hey - I'm busy looking at: " + i + " and " + j);  
4     }  
5 }
```

VS.

```
1 for (int i = 1; i <= Math.pow(2, n); i++) {  
2     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

- What O are these?

$O(n \log n)$

```
1 for (int i = 1; i <= n; i++) {  
2     for(int j = 1; j < n; j = j * 2) {  
3         System.out.println("Hey - I'm busy looking at: " + i + " and " + j);  
4     }  
5 }
```

VS.

$O(2^n)$

```
1 for (int i = 1; i <= Math.pow(2, n); i++) {  
2     System.out.println("Hey - I'm busy looking at: " + i);  
3 }
```

- $O(2^n)$  : recursive fibonacci



$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = F_1 + F_0 = 1 + 0 = 1$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$

$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

$$F_6 = F_5 + F_4 = 5 + 3 = 8$$

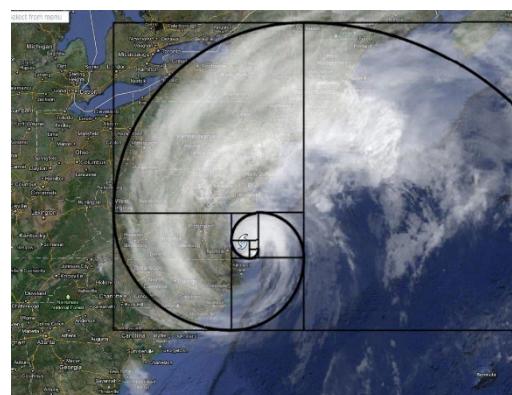
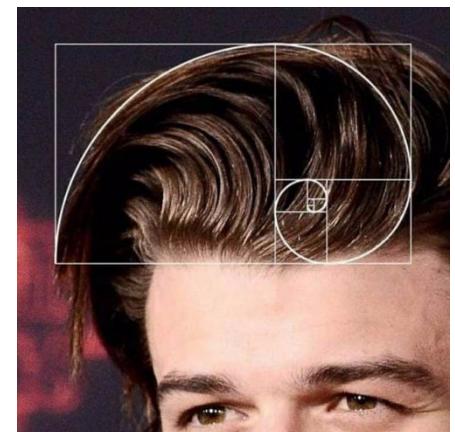
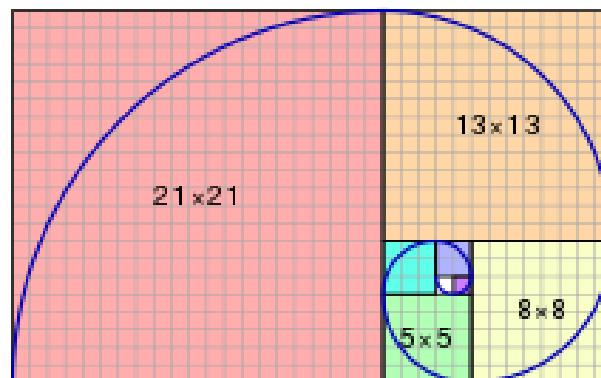
$$F_7 = 13 (8+5)$$

$$F_8 = 21 (13+8)$$

$$F_9 = 34 (21+13)$$

$$F_{10} = 55 (34+21)$$

...



```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Now Fibonacci is defined as

$$F(n) = F(n - 1) + F(n - 2)$$

The characteristic equation for this function will be

$$x^2 = x + 1$$

$$x^2 - x - 1 = 0$$

Solving this by quadratic formula we can get the roots as

$$x = (1 + \sqrt{5})/2 \text{ and } x = (1 - \sqrt{5})/2$$

Now we know that solution of a linear recursive function is given as

$$F(n) = (\alpha_1)^n + (\alpha_2)^n$$

where  $\alpha_1$  and  $\alpha_2$  are the roots of the characteristic equation.

So for our Fibonacci function  $F(n) = F(n - 1) + F(n - 2)$  the solution will be

$$F(n) = ((1 + \sqrt{5})/2)^n + ((1 - \sqrt{5})/2)^n$$

$$O(1.6180)^n$$

For  $F(1)$ , the answer is 1 (the first part of the conditional).

For  $F(n)$ , the answer is  $F(n-1) + F(n-2)$ .

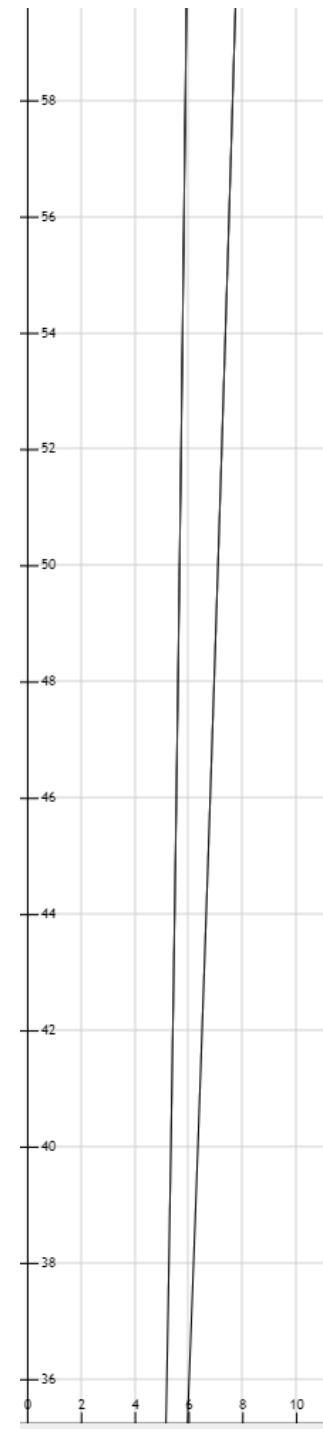
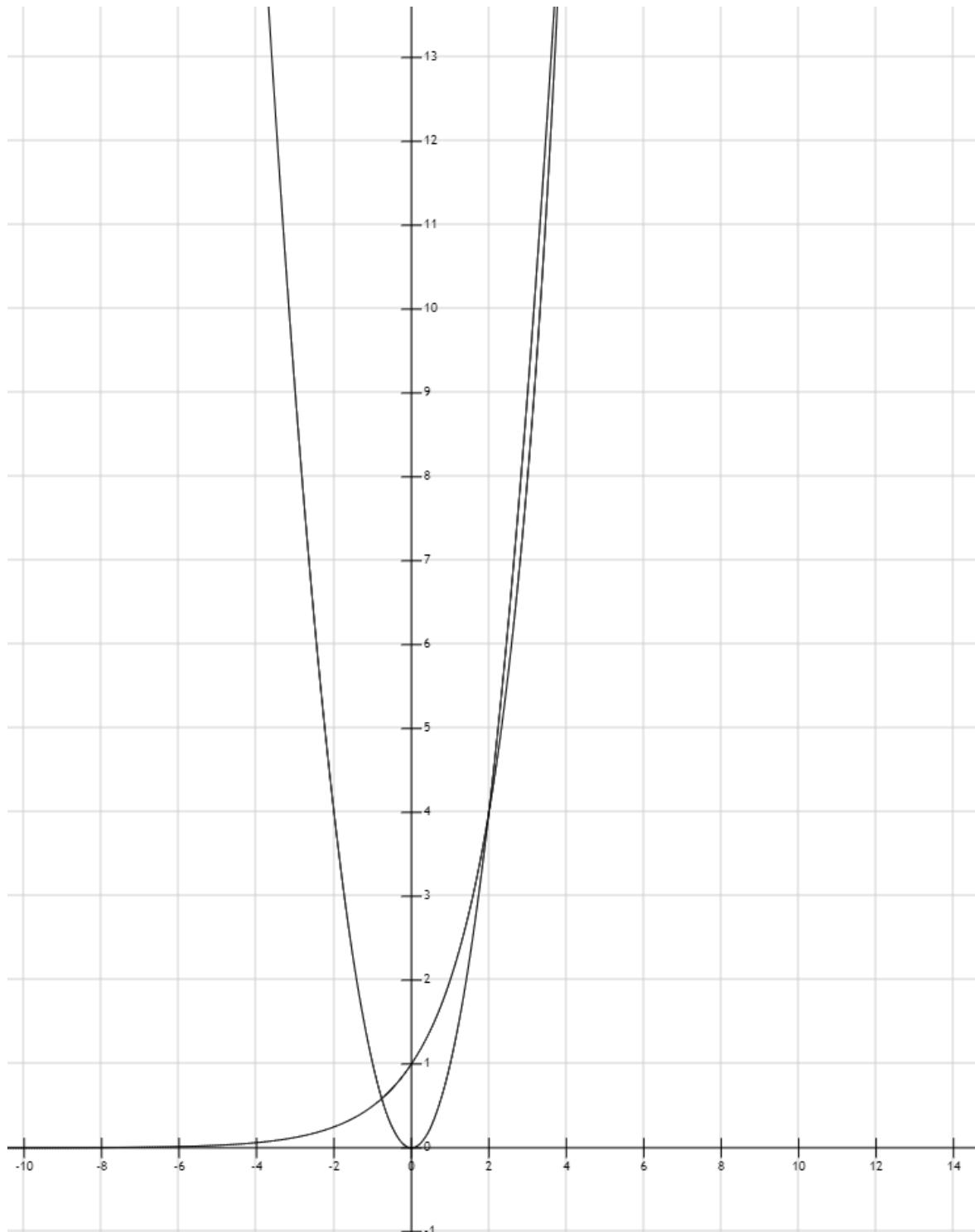
So what function satisfies these rules? Try  $a^n$  ( $a > 1$ ):

$$a^n = a^{(n-1)} + a^{(n-2)}$$

Divide through by  $a^{(n-2)}$ :

$$a^2 = a + 1$$

$O(2^n)$  vs  $O(n^2)$



# Sorting

- Sorting algorithms that are easy to understand (and implement) run in **quadratic time**
- More complicated algorithms cut it to  $O(N \log N)$ 
  - Implementation details are critical to attaining this bound!
- For very specific types of data we can actually do better
  - But we won't study these algorithms extensively

# Selection Sort

## The simplest sorting algorithm

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i;  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ←———— Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ← Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

Look for items less than those in sorted part of array

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i; ← Last item in the sorted part of array  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp; ] Swap the items  
    }  
}
```

Look for items less than those in sorted part of array

What is the complexity?

# Selection sort

```
public static void selectionSort(int[] arr){  
    for(int i=0; i < arr.length-1; i++){  
        int min = i;  
        for(int j=i+1; j < arr.length; j++){  
            if (arr[j] < arr[min])  
                min = j;  
        }  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Selection sort

```
public static void selectionSort(int[] arr){  
  
    for(int i=0; i < arr.length-1; i++){  
  
        int min = i;  
  
        for(int j=i+1; j < arr.length; j++){  
  
            if (arr[j] < arr[min])  
                min = j;  
  
        }  
  
        int temp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = temp;  
    }  
}
```

# Insertion Sort

Good for small **N**

# Insertion sort

Good for small N

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

# Insertion sort

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

What does it look like?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←————
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←———— Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←———— Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←———— Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i]; ←———— Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];      ← Item to be inserted
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;          Insert item
    }
}
```

Until the insertion position is found shift sorted items

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What the complexity of insertion sort  
(in the worst case)?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

# Insertion sort

```
public static void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
```

What if the input array is already sorted?

Best case!

Today...

Margesort  
divide and conquer

But first, merging...

# Merging...

- Say we have two sorted lists, how can we efficiently merge them?

# Merging...

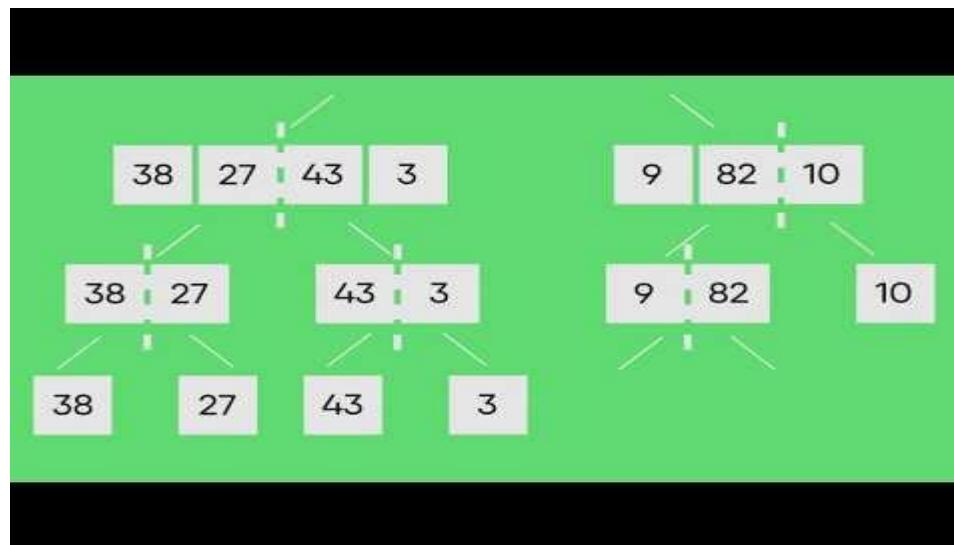
- Say we have two sorted lists, how can we efficiently merge them?

What does it look like?

# Merging...

- Say we have two sorted lists, how can we efficiently merge them?
- **Idea:** Compare the first element in each list to each other, take the smallest and add to the merged list
  - and... repeat

# Our friend:



<https://www.youtube.com/watch?v=JSceec-wEyw>

# Mergesort

1. Divide the array in half
2. Sort the left half
3. Sort the right half
4. Merge the two halves together

What is missing here?

# Mergesort

1. Divide the array in half
2. Sort the left half
3. Sort the right half
4. Merge the two halves together

How do we sort?

What is missing here?

# Mergesort

1. Divide the array in half
2. Sort the left half
3. Sort the right half
4. Merge the two halves together

How do we sort?

Can we avoid sorting?  
How?

What is missing here?

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- 
2. Take the left half, and go back to step 1
  3. Take the right half, and go back to step 1

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- ~~2. Sort the left half~~
- ~~3. Sort the right half~~
2. Take the left half, and go back to step 1 **Until?!**
  3. Take the right half, and go back to step 1 **Until?!**

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- ~~2. Sort the left half~~
- ~~3. Sort the right half~~
2. Take the left half, and go back to step 1 **Until?!**
  3. Take the right half, and go back to step 1 **Until?!**

What does it look like?

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    int mid = (left + right) / 2;
    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    merge(arr, left, mid+1, right);
}
```

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    int mid = (left + right) / 2;
    mergesort(arr, left, mid); ← Divide
    mergesort(arr, mid+1, right); ←
    merge(arr, left, mid+1, right);
}
```

Divide

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    int mid = (left + right) / 2;
    mergesort(arr, left, mid); ← Divide
    mergesort(arr, mid+1, right); ←
    merge(arr, left, mid+1, right); ← Conquer
}
```

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    int mid = (left + right) / 2;
    mergesort(arr, left, mid); ← Divide
    mergesort(arr, mid+1, right); ←
    merge(arr, left, mid+1, right); ← Conquer
}
```

What is missing here?

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int mid = (left + right) / 2;
    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    merge(arr, left, mid+1, right);
}
```

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int mid = (left + right) / 2;
    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    merge(arr, left, mid+1, right);
}
```

What is the complexity of this algorithm?

# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int mid = (left + right) / 2;
    mergesort(arr, left, mid); ← Divide
    mergesort(arr, mid+1, right); ← Divide
    merge(arr, left, mid+1, right);
}
```

What is the complexity of the divide step?

1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$

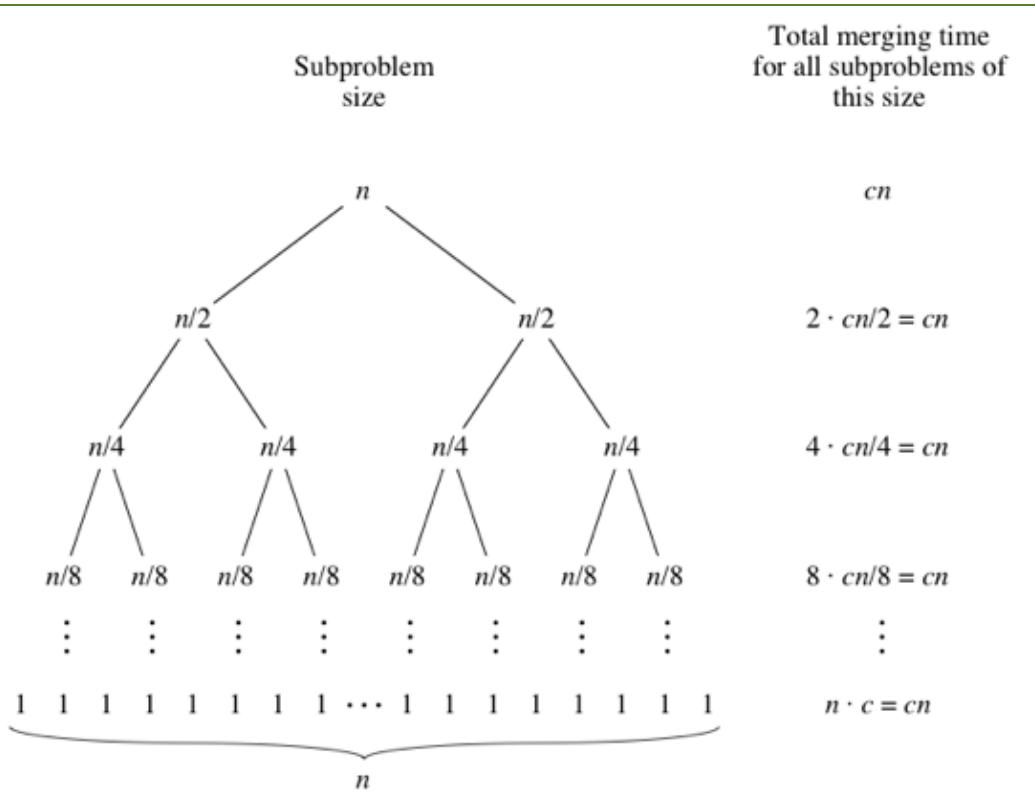
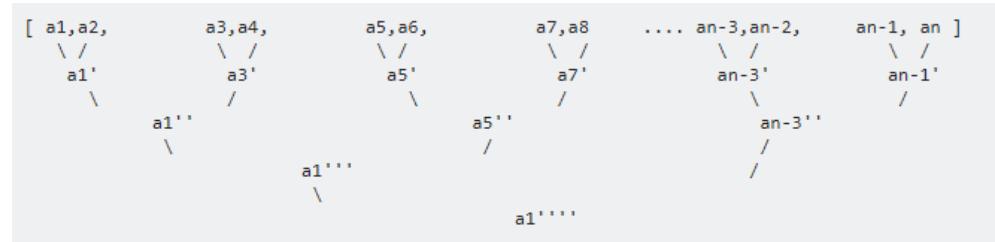
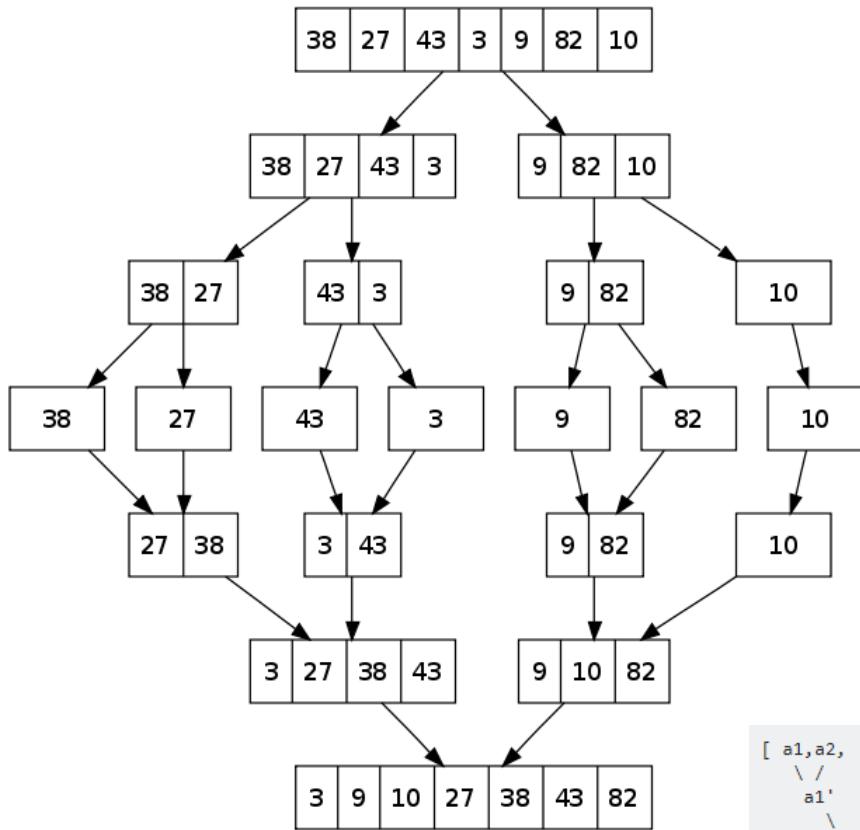
# Mergesort

```
public static void mergesort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int mid = (left + right) / 2;
    mergesort(arr, left, mid); ← Divide
    mergesort(arr, mid+1, right); ← Conquer
    merge(arr, left, mid+1, right); ←
}
```

What is the complexity of the conquerstep?

1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$



1. Dividing step:  $O(1)$  for every step
2. Every merge step takes  $O(n)$
3. Do merge “height of the tree” times, which is  $O(\log n + 1)$

Complexity is:  $(\log n + 1) * (cn)$

$c$  is a constant

So it is  $O(N \log N)$  for best, average, and worst cases.

# Merging...

- Easy concept, tricky coding...
- Lots of special cases:
  - keep track of two indices to step through both arrays (the “front” of each array)
    - *indices do not necessarily move at the same speed*
  - have to stop the loop when either index reaches the end of their array
  - the two arrays are not necessarily the same size
  - what to do when you reach the end of one array but not the other?
  - copy from temp back into the array

```

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int [n1];
    int R[] = new int [n2];

    /*Copy data to temp arrays*/
    for (int i=0; i<n1; ++i)
        L[i] = arr[l + i];
    for (int j=0; j<n2; ++j)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
}

```

```

/* Copy remaining elements of L[] if any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

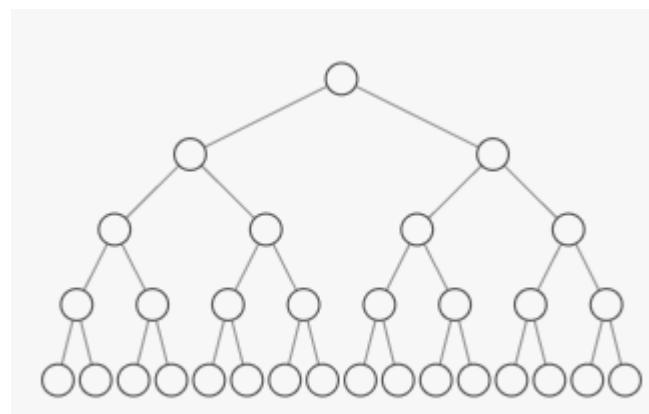
/* Copy remaining elements of R[] if any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

```



# Mergesort caveat!

- The major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array

# Mergesort caveat!

- The major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array
- This means that mergesort requires 2x as much space as the array itself
  - Can be an issue if space is limited!
  - An *in-place* mergesort exists, but is complicated and has worse performance

# Mergesort caveat!

- The major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array
- This means that mergesort requires 2x as much space as the array itself
- To achieve the overall running time of  $O(N \log N)$  it is critical that the running time of the merge phase be linear

| Algorithm      | Time Complexity<br>(worst case) |
|----------------|---------------------------------|
| Selection sort |                                 |
| Insertion sort |                                 |
| Mergesort      |                                 |

| Algorithm      | Time Complexity<br>(worst case) |
|----------------|---------------------------------|
| Selection sort | $O(N^2)$                        |
| Insertion sort | $O(N^2)$                        |
| Mergesort      | $O(N \log N)$                   |

Sorting  
CSC220 | Computer Programming 2

Last Time...

# Selection sort

1. Find the minimum item in the unsorted part of the array
2. Swap it with the first item in the unsorted part of the array
3. Repeat steps 1 and 2 to sort the remainder of the array

# Insertion sort

Good for small N

1. The first array item is the sorted portion of the array
2. Take the second item and insert it in the sorted portion
3. Repeat steps 1 and 2 to sort the remainder of the array

# Mergesort

1. Divide the array in half
2. Sort the left half
3. Sort the right half
4. Merge the two halves together

# Mergesort

1. Divide the array in half
2. Sort the left half
3. Sort the right half
4. Merge the two halves together

How do we sort?

What is missing here?

# Mergesort

1. Divide the array in half
  
2. Sort the left half
3. Sort the right half
  
4. Merge the two halves together

How do we sort?

Can we avoid sorting?  
How?

What is missing here?

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- 
2. Take the left half, and go back to step 1
  3. Take the right half, and go back to step 1

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- 
2. Take the left half, and go back to step 1 Until?!
  3. Take the right half, and go back to step 1 Until?!

# Mergesort

1. Divide the array in half
  2. Sort the left half
  3. Sort the right half
  4. Merge the two halves together
- ~~2. Sort the left half~~
- ~~3. Sort the right half~~

2. Take the left half, and go back to step 1 Until?!
3. Take the right half, and go back to step 1 Until?!

What does it look like?

# Mergesort caveat!

- The major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array
- This means that mergesort requires 2x as much space as the array itself
  - Can be an issue if space is limited!
  - An *in-place* mergesort exists, but is complicated and has worse performance

# Mergesort caveat!

- The major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array
- This means that mergesort requires 2x as much space as the array itself
- To achieve the overall running time of  $O(N \log N)$  it is critical that the running time of the merge phase be linear

| Algorithm      | Time Complexity<br>(worst case) |
|----------------|---------------------------------|
| Selection sort | $O(N^2)$                        |
| Insertion sort | $O(N^2)$                        |
| Mergesort      | $O(N \log N)$                   |

Today...

# Quicksort

another divide and conquer

# Quicksort

1. Select an item in the array to be the *pivot*
2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
3. Sort the left half
4. Sort the right half

# Quicksort

1. Select an item in the array to be the *pivot*
2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
3. Sort the left half
4. Sort the right half

NOTE: after partitioning, the pivot is in it's final position!

# Quicksort

1. Select an item in the array to be the *pivot*
2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
3. Sort the left half
4. Sort the right half

What did you notice?!

4 10 8 7 6 5 3 12 14 2

Choose 6 as the pivot

4 10 8 7 6 5 3 12 14 2

Move the numbers <6 to the left,  
and the numbers >6 to the right of it

4 5 3 2 6 10 8 7 12 14

Choose 5 as the pivot

4 5 3 2 6

Move the numbers <5 to its left,  
and the numbers >5 to the right of it

4 3 2 5 6

4 3 2 5 6

Choose 3 as the pivot

4 3 2 5 6

Move the numbers <3 to the left,  
and the numbers >3 to the right of it

2 3 4 5 6

6 5 3 12 14 2

Choose 7 as the pivot

6 10 8 7 12 14

Move the numbers <7 to the left,  
and the numbers >7 to the right of it

6 7 10 8 12 14

Choose 8 as the pivot

6 7 10 8 12 14

Move the numbers <8 to the left,  
and the numbers >8 to the right of it

6 7 8 10 12 14

6   7    8    10 12 14

Choose 10 as the pivot

6   7    8    10 12 14

Move the numbers <10 to the left,  
and the numbers >10 to the right of it

6   7    8    10 12 14

How do we choose the pivot?

The following example shows selecting the middle number and swap with the last one. So you can also directly choose the last one.

10 7 12 6 3 2 8

Choose the number in the middle of the list as the pivot

10 7 12 6 3 2 8

Move the pivot to the end (swap)

10 7 12 8 3 2 6

↑ The first number that is > than pivot  
↑ The current number

10 7 12 8 3 2 6

Is 10 larger than 6?  
Yes  
So move the blue arrow to the right



Is 7 larger than 6?  
Yes  
So move the blue arrow to the right

10 7 12 8 3 2 6



Is 12 larger than 6?

Yes

So move the blue arrow to the right

10 7 12 8 3 2 6



Is 12 larger than 6?

Yes

So move the blue arrow to the right

10 7 12 8 3 2 6



Is 3 larger than 6?

No

Swap the two numbers of blue  
and yellow arrows

3 7 12 8 10 2 6



Increase the yellow counter

3 7 12 8 10 2 6



Increase the blue counter

3 7 12 8 10 2 6



Is 2 larger than 6?

No

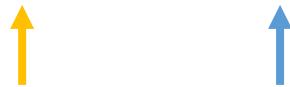
Swap the two numbers of blue  
and yellow arrows

3 2 12 8 10 7 6



Increase the yellow counter

3 2 12 8 10 7 6



Swap the pivot to the number pointed by the yellow pointer

3 2 6 8 10 7 12



# Quicksort

1. Select an item in the array to be the *pivot*
  2. ***Partition*** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
  3. Sort the left half
  4. Sort the right half
- ~~3. Sort the left half~~
- ~~4. Sort the right half~~
3. Take the left half, and go back to step 1
  4. Take the right half, and go back to step 1

# Quicksort

1. Select an item in the array to be the *pivot*
  2. ***Partition*** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
  3. Sort the left half
  4. Sort the right half
- ~~3. Sort the left half~~
- ~~4. Sort the right half~~
3. Take the left half, and go back to step 1 Until?!
  4. Take the right half, and go back to step 1 Until?!

# Quicksort

1. Select an item in the array to be the *pivot*
  2. ***Partition*** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
  3. Sort the left half
  4. Sort the right half
- ~~3. Sort the left half~~
- ~~4. Sort the right half~~
3. Take the left half, and go back to step 1 Until?!
  4. Take the right half, and go back to step 1 Until?!

What does it look like?

```
public static void quicksort(int[] arr, int left, int right)
{
    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

```
public static void quicksort(int[] arr, int left, int right)
{
    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

What are we missing?

```
public static void quicksort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

```
public static void quicksort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

What is the divide step?

```
public static void quicksort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

What is the divide step?  
What is the conquer step?

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array
- Mergesort requires an extra, temporary array for merging

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array
- Mergesort requires an extra, temporary array for merging
- However, in-place partitioning for quicksort requires some careful thought...

# In-place partitioning

1. select an item in the array to be the ***pivot***
2. swap the pivot with the last item in the array (*just get it out of the way*)
3. step from left to right until we find an item > pivot
  - *this item needs to be on the **right** of the partition*
4. step from right to left until we find an item < pivot
  - *this item needs to be on the **left** of the partition*
5. swap items
6. continue until left and right stepping cross
7. swap pivot with left stepping item

# In-place partitioning

1. select an item in the array to be the ***pivot***
2. swap the pivot with the last item in the array (*just get it out of the way*)
3. step from left to right until we find an item > pivot
  - *this item needs to be on the **right** of the partition*
4. step from right to left until we find an item < pivot
  - *this item needs to be on the **left** of the partition*
5. swap items
6. continue until left and right stepping cross
7. swap pivot with left stepping item

What does this look like?

```
// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
       places the pivot element at its correct
       position in sorted array, and places all
       smaller (smaller than pivot) to left of
       pivot and all greater elements to right
       of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than the pivot
            if (arr[j] < pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }

    /* The main function that implements QuickSort()
       arr[] --> Array to be sorted,
       low --> Starting index,
       high --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
               now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

# Choosing a pivot

- What is the best choice?

# Choosing a pivot

- The median of all array items is the best possible choice
  - why?

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$
- It is important that we avoid the worst case

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$
- It is important that we avoid the worst case
  - What IS the worst case(s)?

# Choosing a pivot

- Any nonrandom pivot selection has some devious input that causes  $O(N^2)$
- Trade-off between quality of pivot and time to select
- Selection cost should always be  $O(c)$ 
  - i.e., it should not depend on  $N!$

# Quicksort complexity

- Performance of quick sort heavily depends on which array item is chosen as the pivot
- **Best case:** pivot partitions the array into two equally- sized subarrays *at each stage*
- **Worst case:** partition generates an empty subarray *at each stage*

# Quicksort complexity

- Performance of quick sort heavily depends on which array item is chosen as the pivot
- **Best case:** pivot partitions the array into two equally- sized subarrays *at each stage*

$O(N \log N)$

- **Worst case:** partition generates an empty subarray *at each stage*

$O(N^2)$

# Quicksort vs Mergesort

- Both are divide and conquer algorithms (recursive)
- Mergesort sorts “on the way up”
  - after the base case is reached, sorting is done as the calls return and merge
- Quicksort sorts “on the way down”
  - once the base case is reached, that part of the array is sorted

- Mergesort is also  $O(N \log N)$  in the *worst* case
  - so, why not always use mergesort?

- Mergesort is also  $O(N \log N)$  in the *worst* case
  - so, why not always use mergesort?
- Mergesort requires  $2N$  space
  - and, copying everything from the merged array back to the original takes time
- Quicksort requires no extra space
  - thus, no copying overhead!
  - but, in  $O(N^2)$  worst case

# Sorting summary

Worst

Selection sort

$O(N^2)$

Insertion sort

$O(N^2)$

Mergesort

$O(N \log N)$

Quicksort

$O(N^2)$

# Sorting summary

|                | <b>Best</b>   | <b>Worst</b>  |
|----------------|---------------|---------------|
| Selection sort | $O(N^2)$      | $O(N^2)$      |
| Insertion sort | $O(N)$        | $O(N^2)$      |
| Mergesort      | $O(N \log N)$ | $O(N \log N)$ |
| Quicksort      | $O(N \log N)$ | $O(N^2)$      |

# Sorting summary

|                | Best          | Worst         | Note                  |
|----------------|---------------|---------------|-----------------------|
| Selection sort | $O(N^2)$      | $O(N^2)$      | Not used in practice! |
| Insertion sort | $O(N)$        | $O(N^2)$      | No comment!           |
| Mergesort      | $O(N \log N)$ | $O(N \log N)$ | 2x space overhead     |
| Quicksort      | $O(N \log N)$ | $O(N^2)$      | Depends on pivot      |

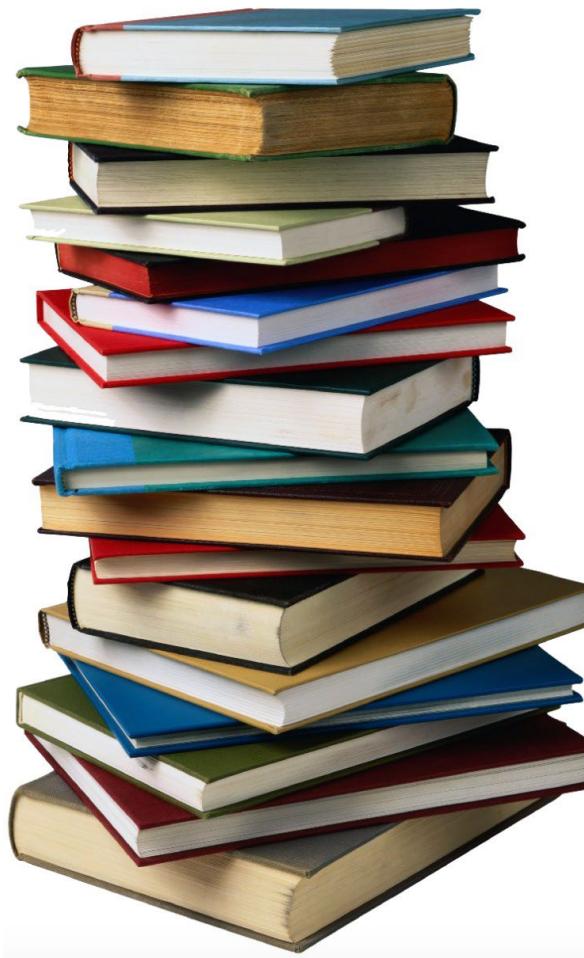
Next Time...

- Stack and queues
  - Reading: Chapter 14
- Start on your assignment early
- Read the book chapter **before** and **after** session
- Start preparing for the midterm exam!

# Queue

## CSC220|Computer Programming 2

# Stacks



- A **stack** is a data structure in which insertion and removal is restricted to the **top** (or end) of the list
- Also called FIRST-IN, LAST-OUT (FILO)
  - Insertion always adds an item to the end
  - Deletion always removes an item from the end

# Important methods

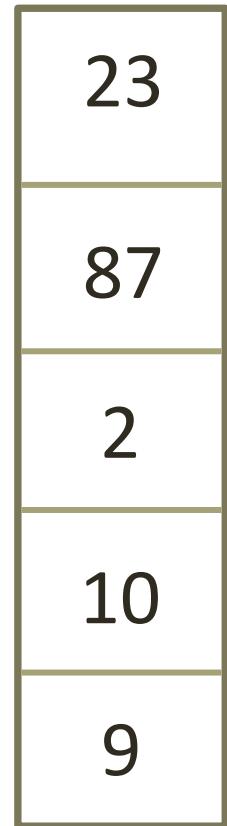
- `push`
  - Inserts an item on to the top of the stack
- `pop`
  - Removes and returns the item on the top of the stack
- `peek`
  - Returns but does not remove the top of the stack

# Important methods

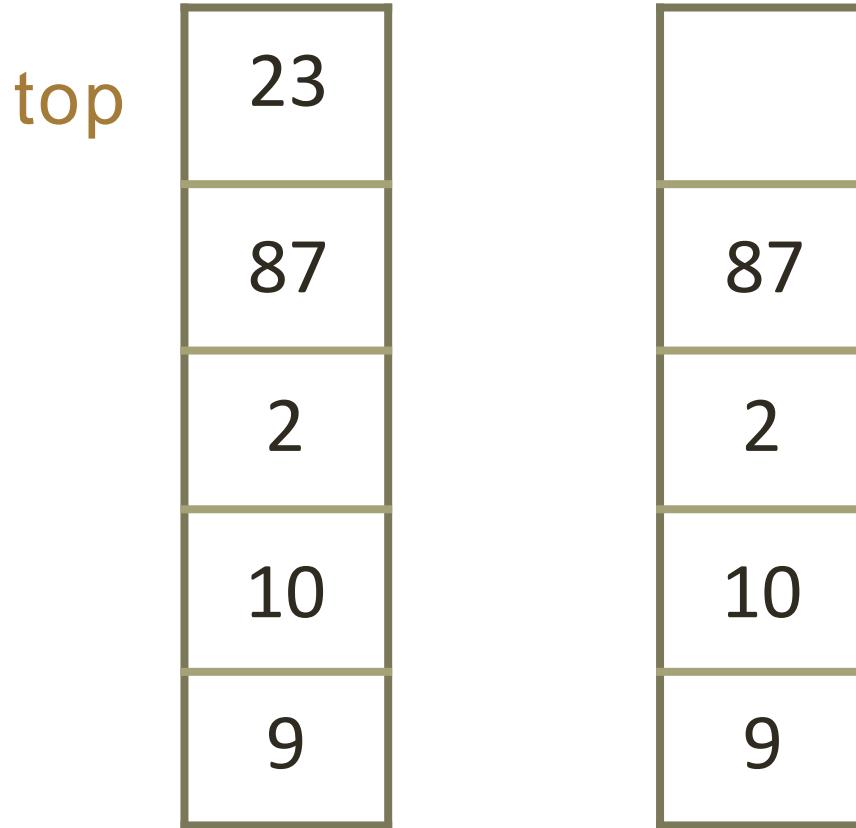
- push
  - Inserts an item on to the top of the stack
- pop
  - Removes and returns the item on the top of the stack
- peek
  - Returns but does not remove the top of the stack

Consecutive calls to pop will return items in the reverse order that they were pushed

top



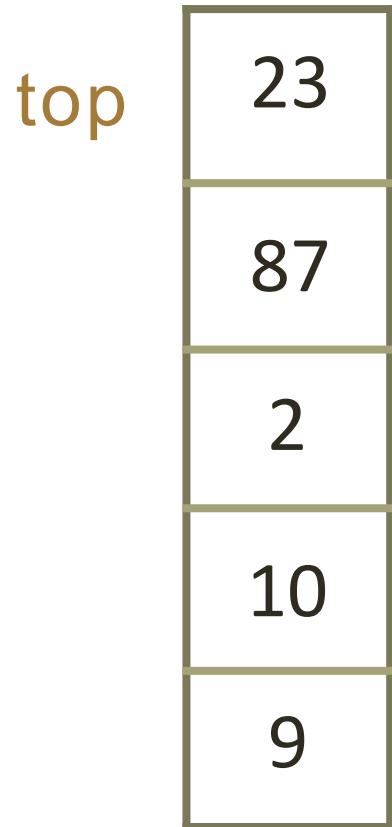
`pop();`



It is useful to think of stacks as standing upright (like a stack of books!)

```
pop();
```

```
push(5);
```



It is useful to think of stacks as standing upright (like a stack of books!)

Today...

# Queue



- A **queue** is a FIRST-IN, FIRST-OUT data structure
  - FIFO
- Insert on the back, remove from the front
- Like a stack, all operations are  $O(1)$

- A **queue** is a FIRST-IN, FIRST-OUT data structure
  - FIFO
- Operations:
  - *enqueue*... adds an item to the **back** of the queue
    - Also “offer”
  - *dequeue*... removes and returns the item at the **front**
    - Also “poll”

front



back

enqueue (8)



enqueue (8)

dequeue ()



enqueue (8)

dequeue ()

enqueue (7)



enqueue (8)

dequeue ()

enqueue (7)



How can we implement a QUEUE so that all operations are guaranteed to be O(1)?

# As an array....

- Keep track of front and back indices
- front and back advance through the array
  - *enqueueing* advances back
  - *dequeueing* advance front



# As an array....

- Keep track of front and back indices
- front and back advance through the array
  - *enqueueing* advances back
  - *dequeueing* advance front



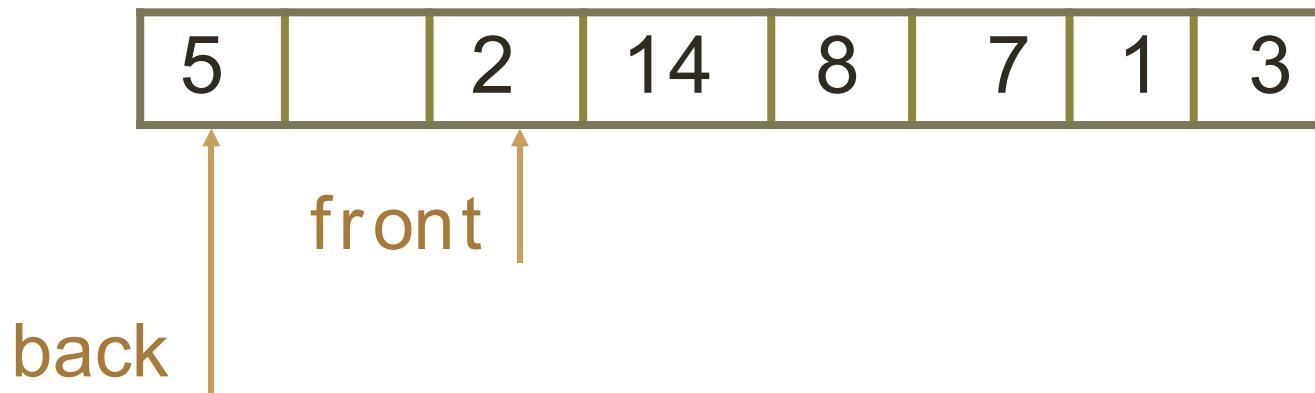
- What happens when back reaches the end of the array?



enqueue (5)



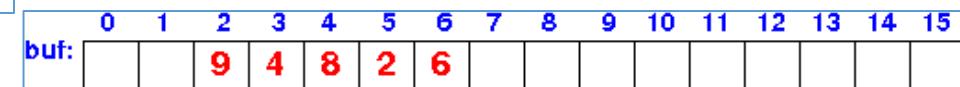
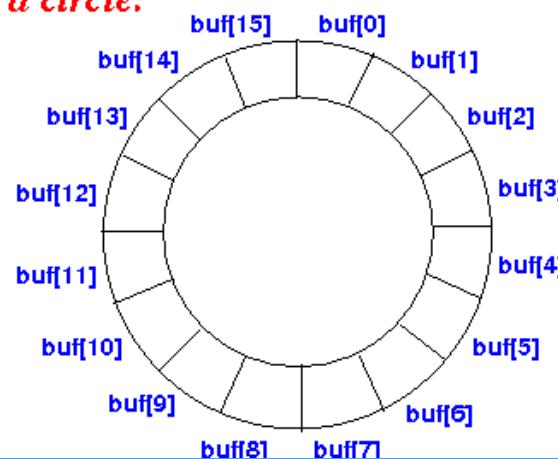
enqueue (5)



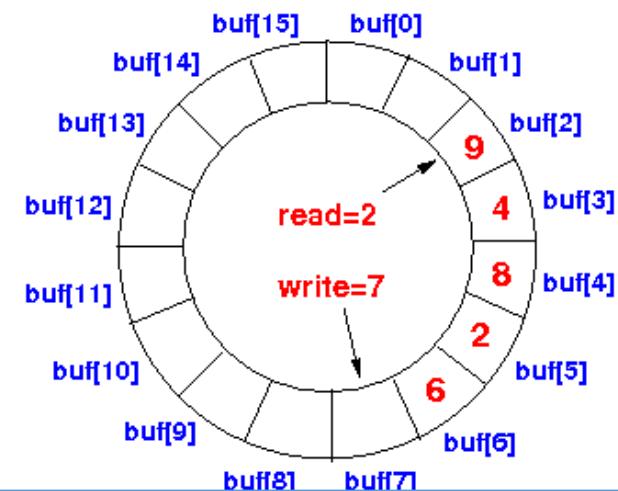
## Array:



Pretend array is a circle:



read=2  
write=7



```
1 // Java program for array implementation of queue
2
3 // A class to represent a queue
4 class Queue
5 {
6     int front, rear, size;
7     int capacity;
8     int array[];
9
10    public Queue(int capacity) {
11        this.capacity = capacity;
12        front = this.size = 0;
13        rear = capacity - 1;
14        array = new int[this.capacity];
15    }
16
17    // Queue is full when size becomes equal to
18    // the capacity
19    boolean isFull(Queue queue)
20    { return (queue.size == queue.capacity); }
21
22    // Queue is empty when size is 0
23    boolean isEmpty(Queue queue)
24    { return (queue.size == 0); }
25
26    // Method to add an item to the queue.
27    // It changes rear and size
28    void enqueue( int item)
29    {
30        if (isFull(this))
31            return;
32        this.rear = (this.rear + 1)%this.capacity;
33        this.array[this.rear] = item;
34        this.size = this.size + 1;
35        System.out.println(item+ " enqueued to queue");
36    }
37
38
39
40    // Method to remove an item from queue.
41    // It changes front and size
42    int dequeue()
43    {
44        if (isEmpty(this))
45            return Integer.MIN_VALUE;
46
47        int item = this.array[this.front];
48        this.front = (this.front + 1)%this.capacity;
49        this.size = this.size - 1;
50        return item;
51    }
52
53    // Method to get front of queue
54    int front()
55    {
56        if (isEmpty(this))
57            return Integer.MIN_VALUE;
58
59        return this.array[this.front];
60    }
61
62    // Method to get rear of queue
63    int rear()
64    {
65        if (isEmpty(this))
66            return Integer.MIN_VALUE;
67
68        return this.array[this.rear];
69    }
70
71 }
```



# Performance

- Using wrap-around, all operations are  $O(1)$  on average

# Performance

- Using wrap-around, all operations are  $O(1)$  on average
- $O(N)$  array growing is still a problem in the worst case!

# Performance

- Using wrap-around, all operations are  $O(1)$  on average
- $O(N)$  array growing is still a problem in the worst case!
- How do we handle array growth if there is wrap-around in the queue?
  - This is non-trivial...

# Priority Queue

- Like a queue, but items returned in order of ***priority***
  - *Dequeue* operation always returns the item with the **highest priority**
  - If two items have the same priority, the first one in the queue is returned

- Like a queue, but items returned in order of ***priority***
  - *Dequeue* operation always returns the item with the highest priority
  - If two items have the same priority, the first one in the queue is returned

Java.util.PriorityQueue class in Java. It is a priority queue based on priority heap.

- Elements in this class are in natural order or depends on the Constructor we used at the time of construction.
- It doesn't allow inserting a non-comparable object, if it relies on natural ordering.

## **Constructors:**

- PriorityQueue():** Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
- PriorityQueue(Collection c):** Creates a PriorityQueue containing the elements in the specified collection.
- PriorityQueue(int initialCapacity):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
- PriorityQueue(int initialCapacity, Comparator comparator):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
- PriorityQueue(PriorityQueue c):** Creates a PriorityQueue containing the elements in the specified priority queue.
- PriorityQueue(SortedSet c):** Creates a PriorityQueue containing the elements in the specified sorted set.

## Methods:

1. **add(element) : java.util.PriorityQueue.add()** insert the elements to the Priority Queue.

### Syntax :

```
public boolean add(E e)
Parameters :
element : the element we need to add.
Return :
call return true.
Exception :
-&gt; ClassCastException
-&gt; NullPointerException
```

2. **comparator() : java.util.PriorityQueue.comparator()** orders the elements in the queue.

### Syntax :

```
public Comparator comparator()
Parameters :
-----
Return :
orders the queue or return null, if it is naturally ordered
Exception :
-----
```

3. **contains(Object obj) : java.util.PriorityQueue.contains(obj)** returns true if the priority queue contains the element "obj".

### Syntax :

```
public boolean contains(Object obj)
Parameters :
obj : object to be checked
Return :
true - if the object is present else, return false
Exception :
```

5. **offer(element) : java.util.PriorityQueue.offer()** is required to insert a specific element to the given priority queue.

Syntax :

```
public boolean offer(E element)
Parameters :
element : specific element to be entered.
Return :
call return true.
Exception :
-&gt; ClassCastException
-&gt; NullPointerException
```

6. **peek() : java.util.PriorityQueue.peek()** identifies the head element of the queue.

Syntax :

```
public E peek()
Parameters :
-----
Return :
calls if head exists, else null
Exception :
-----
```

7. **poll() : java.util.PriorityQueue.poll()** identifies the head and then removes it.

Syntax :

```
public E poll()
Parameters :
---
Return :
calls if head exists, else null
Exception :
-----
```

```
1 // Java program to demonstrate working of priority queue in Java
2 import java.util.*;
3
4 class Example
5 {
6     public static void main(String args[])
7     {
8         // Creating empty priority queue
9         PriorityQueue<String> pQueue =
10             new PriorityQueue<String>();
11
12         // Adding items to the pQueue using add()
13         pQueue.add("C");
14         pQueue.add("C++");
15         pQueue.add("Java");
16         pQueue.add("Python");
17
18         // Printing the most priority element
19         System.out.println("Head value using peek function:" +
20                             + pQueue.peek());
21
22         // Printing all elements
23         System.out.println("The queue elements:");
24         Iterator itr = pQueue.iterator();
25         while (itr.hasNext())
26             System.out.println(itr.next());
27
28         // Removing the top priority element (or head) and
29         // printing the modified pQueue using poll()
30         pQueue.poll();
31         System.out.println("After removing an element" +
32                           "with poll function:");
33         Iterator<String> itr2 = pQueue.iterator();
34         while (itr2.hasNext())
35             System.out.println(itr2.next());
```

```
36
37     // Removing Java using remove()
38     pQueue.remove("Java");
39     System.out.println("after removing Java with" +
40                         " remove function:");
41     Iterator<String> itr3 = pQueue.iterator();
42     while (itr3.hasNext())
43         System.out.println(itr3.next());
44
45     // Check if an element is present using contains()
46     boolean b = pQueue.contains("C");
47     System.out.println ("Priority queue contains C " +
48                         "or not?: " + b);
49
50     // Getting objects from the queue using toArray()
51     // in an array and print the array
52     Object[] arr = pQueue.toArray();
53     System.out.println ("Value in array: ");
54     for (int i = 0; i<arr.length; i++)
55         System.out.println ("Value: " + arr[i].toString()) ;
56 }
57 }
58 }
```

Next Time...

- Next week
  - Linked List TUESDAY
    - Chapter 16
  - Test THURSDAY

Stacks

CSC220 | Computer Programming 2

Last Time...

# Sorting

# Sorting algorithms we saw

- Selection sort
- Insertion sort
- Mergesort
- Quicksort

# Quicksort

another divide and conquer

# Quicksort

1. Select an item in the array to be the *pivot*
2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
3. Sort the left half
4. Sort the right half

# Quicksort

1. Select an item in the array to be the *pivot*
2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
3. Sort the left half
4. Sort the right half

# Quicksort

1. Select an item in the array to be the *pivot*
  2. *Partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right
  3. Sort the left half
  4. Sort the right half
- ~~3. Sort the left half~~
- ~~4. Sort the right half~~
3. Take the left half, and go back to step 1
  4. Take the right half, and go back to step 1

```
public static void quicksort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index-1);
    quicksort(arr, pivot_index+1, right);
}
```

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array
- Mergesort requires an extra, temporary array for merging

# Quicksort

- A huge benefit of quicksort is that it can be done **in-place**
  - i.e., you can do the sort within the original array
- Mergesort requires an extra, temporary array for merging
- However, in-place partitioning for quicksort requires some careful thought...

# Choosing a pivot

- What is the best choice?

# Choosing a pivot

- The median of all array items is the best possible choice
  - why?

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$
- It is important that we avoid the worst case

# Choosing a pivot

- The median of all array items is the best possible choice
  - Is time-consuming to compute
  - Finding true median is  $O(N)$
- It is important that we avoid the worst case
  - What IS the worst case(s)?

# Choosing a pivot

- Any nonrandom pivot selection has some devious input that causes  $O(N^2)$
- Trade-off between quality of pivot and time to select
- Selection cost should always be  $O(c)$ 
  - i.e., it should not depend on  $N!$

# Quicksort complexity

- Performance of quick sort heavily depends on which array item is chosen as the pivot
- **Best case:** pivot partitions the array into two equally- sized subarrays *at each stage*
- **Worst case:** partition generates an empty subarray *at each stage*

# Quicksort complexity

- Performance of quick sort heavily depends on which array item is chosen as the pivot
- **Best case:** pivot partitions the array into two equally-sized subarrays *at each stage*

$O(N \log N)$

- **Worst case:** partition generates an empty

subarray *at each stage*

$O(N^2)$

# Quicksort vs Mergesort

- Both are divide and conquer algorithms (recursive)
- Mergesort sorts “on the way up”
  - after the base case is reached, sorting is done as the calls return and merge
- Quicksort sorts “on the way down”
  - once the base case is reached, that part of the array is sorted

- Mergesort is also  $O(N \log N)$  in the *worst* case
  - so, why not always use mergesort?

- Mergesort is also  $O(N \log N)$  in the *worst* case
  - so, why not always use mergesort?
- Mergesort requires  $2N$  space
  - and, copying everything from the merged array back to the original takes time
- Quicksort requires no extra space
  - thus, no copying overhead!
  - but, in  $O(N^2)$  worst case

# Sorting summary

Worst

Selection sort

$O(N^2)$

Insertion sort

$O(N^2)$

Mergesort

$O(N \log N)$

Quicksort

$O(N^2)$

# Sorting summary

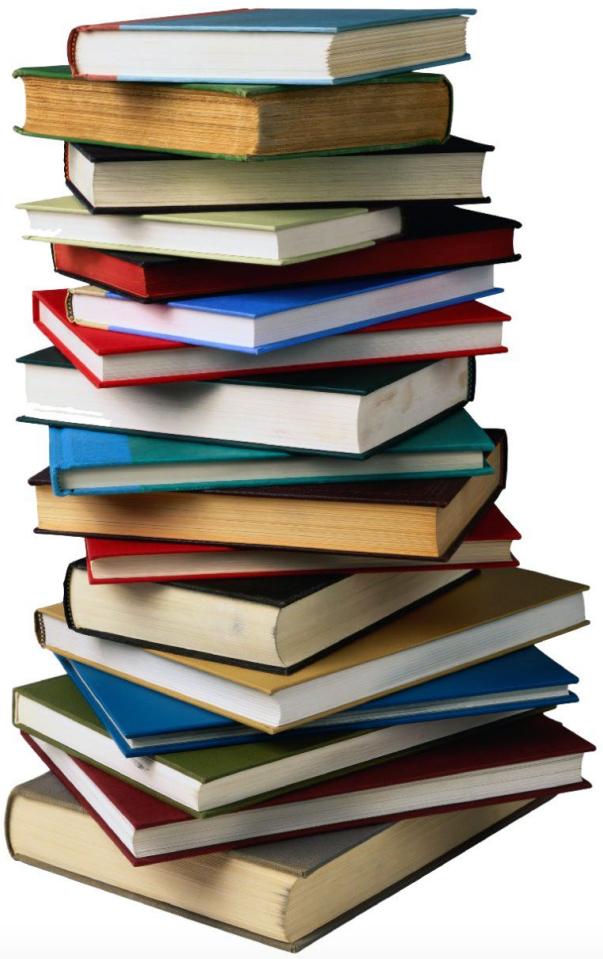
|                | <b>Best</b>   | <b>Worst</b>  |
|----------------|---------------|---------------|
| Selection sort | $O(N^2)$      | $O(N^2)$      |
| Insertion sort | $O(N)$        | $O(N^2)$      |
| Mergesort      | $O(N \log N)$ | $O(N \log N)$ |
| Quicksort      | $O(N \log N)$ | $O(N^2)$      |

# Sorting summary

|                | Best          | Worst         | Note                  |
|----------------|---------------|---------------|-----------------------|
| Selection sort | $O(N^2)$      | $O(N^2)$      | Not used in practice! |
| Insertion sort | $O(N)$        | $O(N^2)$      | No comment!           |
| Mergesort      | $O(N \log N)$ | $O(N \log N)$ | 2x space overhead     |
| Quicksort      | $O(N \log N)$ | $O(N^2)$      | Depends on pivot      |

Today...

# Stacks



# Basic idea

- Store an ordered sequence of values with a minimal set of operations
  - Put values into it (add)
  - Take values out (remove)
  - Test whether there is any values left (is empty)

# Basic idea

- Store an ordered sequence of values with a minimal set of operations
  - Put values into it (add)
  - Take values out (remove)
  - Test whether there is any values left (is empty)

Stack<E>

# Basic idea

- Store an ordered sequence of values with a minimal set of operations
  - Put values into it (add)
  - Take values out (remove)
  - Test whether there is any values left (is empty)

Stack<E>

Stack<String>

- A **stack** is a data structure in which insertion and removal is restricted to the **top** (or end) of the list

- A **stack** is a data structure in which insertion and removal is restricted to the **top** (or end) of the list
- Also called FIRST-IN, LAST-OUT (FILO)
  - Insertion always adds an item to the end
  - Deletion always removes an item from the end

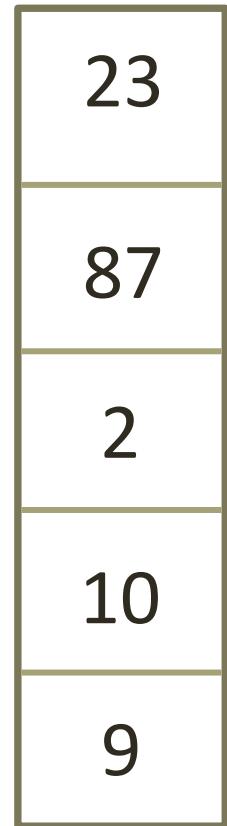
# Important methods

- **push**
  - Inserts an item on to the top of the stack
- **pop**
  - Removes and returns the item on the top of the stack
- **peek**
  - Returns but does not remove the top of the stack

# Important methods

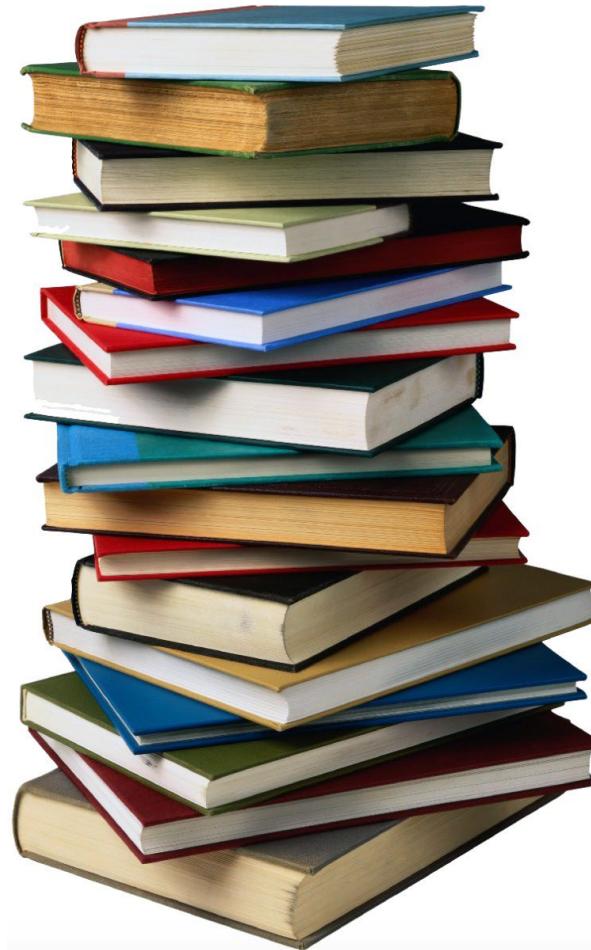
- **push**
  - Inserts an item on to the top of the stack
- **pop**
  - Removes and returns the item on the top of the stack
- **peek**
  - Returns but does not remove the top of the stack
- **Consecutive calls to pop will return items in the reverse order that they were pushed**

top



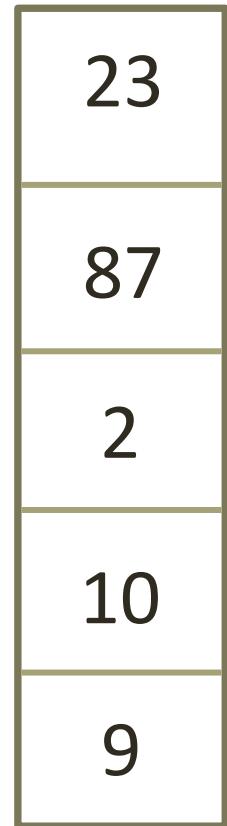
top

|    |
|----|
| 23 |
| 87 |
| 2  |
| 10 |
| 9  |

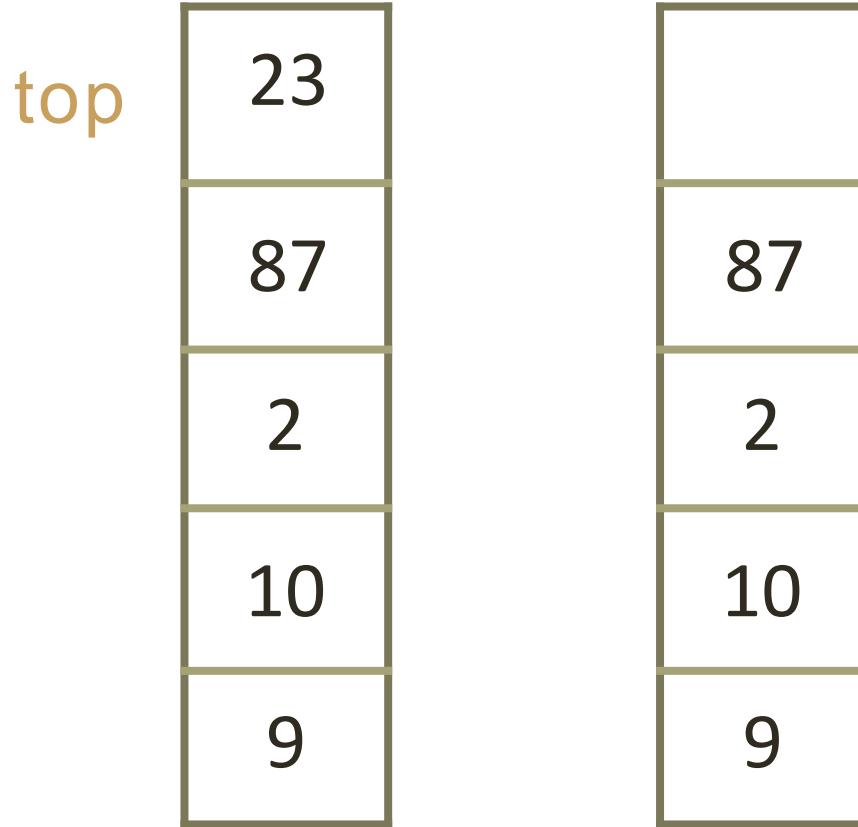


It is useful to think of stacks as standing upright (like a stack of books!)

top



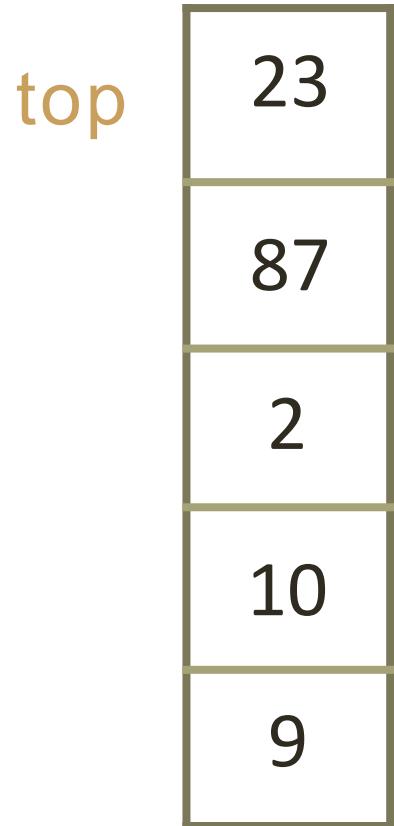
`pop();`



It is useful to think of stacks as standing upright (like a stack of books!)

`pop();`

`push(5);`



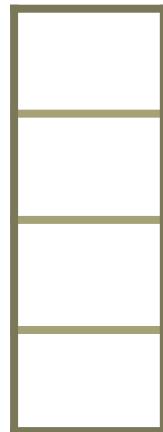
It is useful to think of stacks as standing upright (like a stack of books!)

# Performance

- push, pop, and peek **must** all be  $O(1)$
- How can we implement a stack so that all 3 operations are guaranteed to be  $O(1)$ ?
- We need a very efficient data structure if we expect to only access the last element

As an array...

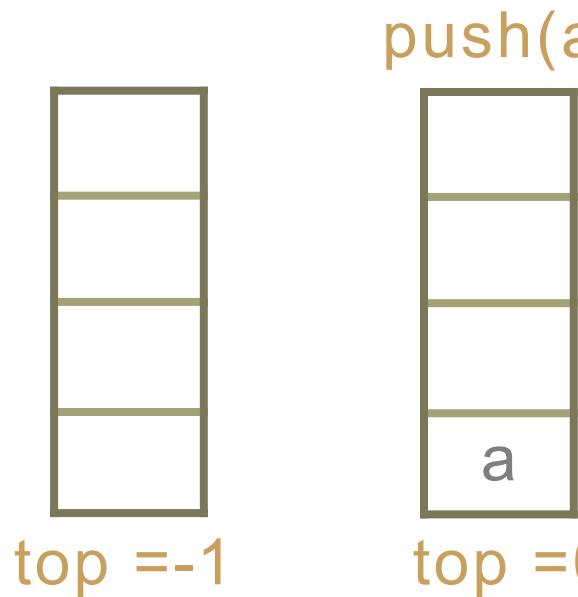
- NOTE: keep track of a `top` index
- To `push`, increment `top`, then add the item at that index
- To `pop`, return the item at index `top`, and decrement `top`



`top = -1`

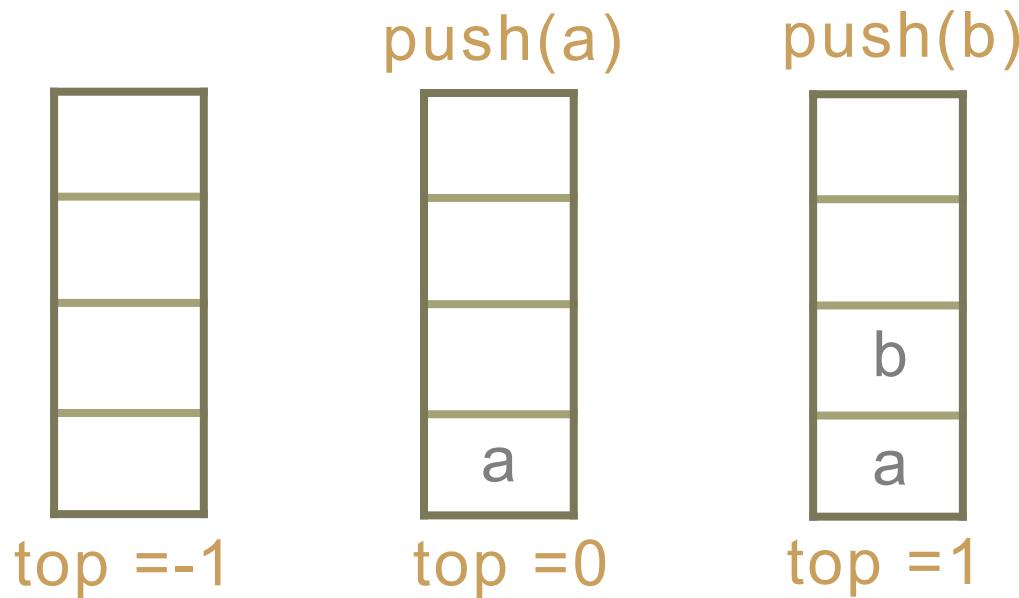
# As an array...

- NOTE: keep track of a `top` index
- To `push`, increment `top`, then add the item at that index
- To `pop`, return the item at index `top`, and decrement `top`



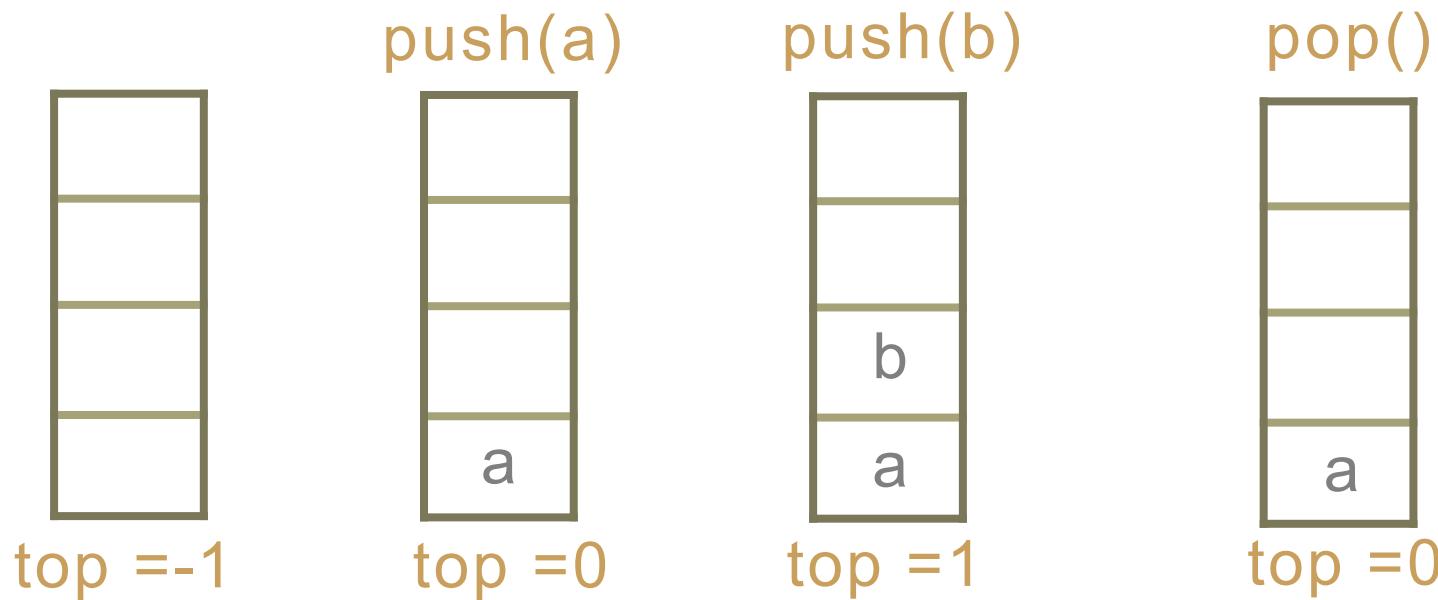
# As an array...

- NOTE: keep track of a `top` index
- To `push`, increment `top`, then add the item at that index
- To `pop`, return the item at index `top`, and decrement `top`



# As an array...

- NOTE: keep track of a `top` index
- To `push`, increment `top`, then add the item at that index
- To `pop`, return the item at index `top`, and decrement `top`



# Performance

- If we try to **push** when the underlying array is full, the array must be grown
- Any **push** that requires resizing the array takes  **$O(N)$**  time
- All other operations are constant,  **$O(1)$**
- Since **pushes** that resize the array are rare, the average case for push is still  **$O(1)$**

```
1 import java.util.*;
2
3 class Stack
4 {
5     private int arr[];
6     private int top;
7     private int capacity;
8
9     // Constructor to initialize stack
10    Stack(int size)
11    {
12        arr = new int[size];
13        capacity = size;
14        top = -1;
15    }
16
17    // Utility function to add an element x in the stack
18    public void push(int x)
19    {
20        if (isFull())
21        {
22            System.out.println("OverFlow\nProgram Terminated\n");
23            System.exit(1);
24        }
25
26        System.out.println("Inserting " + x);
27        arr[++top] = x;
28    }
}
```

```
29
30     // Utility function to pop top element from the stack
31     public int pop()
32     {
33         // check for stack underflow
34         if (isEmpty())
35         {
36             System.out.println("UnderFlow\nProgram Terminated");
37             System.exit(1);
38         }
39
40         System.out.println("Removing " + peek());
41
42         // decrease stack size by 1 and (optionally) return the popped element
43         return arr[top--];
44     }
45
46     // Utility function to return top element in a stack
47     public int peek()
48     {
49         if (!isEmpty())
50             return arr[top];
51         else
52             System.exit(1);
53
54         return -1;
55     }
56 }
```

```
56
57     // Utility function to return the size of the stack
58     public int size()
59     {
60         return top + 1;
61     }
62
63     // Utility function to check if the stack is empty or not
64     public Boolean isEmpty()
65     {
66         return top == -1;      // or return size() == 0;
67     }
68
69     // Utility function to check if the stack is full or not
70     public Boolean isFull()
71     {
72         return top == capacity - 1;    // or return size() == capacity;
73     }
74
75     public static void main (String[] args)
76     {
77         Stack stack = new Stack(3);
78
79         stack.push(1);          // Inserting 1 in the stack
80         stack.push(2);          // Inserting 2 in the stack
81
82         stack.pop();           // removing the top 2
83         stack.pop();           // removing the top 1
84
85         stack.push(3);          // Inserting 3 in the stack
86
87         System.out.println("Top element is: " + stack.peek());
88         System.out.println("Stack size is " + stack.size());
89
90         stack.pop();           // removing the top 3
91
92         // check if stack is empty
93         if (stack.isEmpty())
94             System.out.println("Stack Is Empty");
95         else
96             System.out.println("Stack Is Not Empty");
97     }
98 }
```

# Postfix Notation

- We usually see expressions written in **infix notation**
- Place an *operator* in between a left and right *operand*

a + b

- The order of operations is not clear from the expression without parentheses
  - although, left-to-right is often assumed

1 + 2 \* 3 = ?

- We usually see expressions written in **infix notation**
- Place an *operator* in between a left and right *operand*

a + b

- The order of operations is not clear from the expression without parentheses
    - although, left-to-right is often assumed
- 1 + 2 \* 3 = ?
- *answer is 7, but some calculators will give 9!*

# Postfix expressions

- A syntax lacking parentheses that can be parsed without ambiguity
  - Also called *reverse polish notation*
- Two operands, followed by an operator

a b +

# Postfix expressions

1 2 3 \* +

# Postfix expressions

1 2 3 \* +  
1 6 +

The diagram illustrates the evaluation of a postfix expression. The input expression is "1 [2 3 \*] +". A sub-expression "[2 3 \*]" is highlighted with a brown border. An arrow points from this sub-expression to the output expression "1 6 +", where the value "6" is highlighted in brown. This visualizes how the stack grows during the evaluation of the postfix expression.

# Postfix expressions

$$\begin{array}{r} 1 \boxed{2 \ 3 \ * \ +} \\ 1 \ 6 \ + \longrightarrow 7 \end{array}$$

A diagram illustrating the evaluation of a postfix expression. The top row shows the expression "1 2 3 \* +". The tokens "2", "3", and "\*" are enclosed in a brown rectangular box. A brown arrow points from this box down to the bottom row. The bottom row shows the expression "1 6 +". A brown arrow points from the right side of the "+" operator to the number "7", indicating the result of the evaluation.

# Postfix expressions

$$\begin{array}{r} 1 \boxed{2 \ 3 \ * \ +} \\ 1 \ 6 \ + \longrightarrow 7 \end{array}$$

$$1 \ 2 \ 3 \ * \ + \ 4 \ - \longrightarrow ?$$

# Postfix expressions

$$\begin{array}{r} 1 \boxed{2 \ 3 \ * \ +} \\ 1 \ 6 \ + \longrightarrow 7 \end{array}$$

$$1 \ 2 \ 3 \ * \ + \ 4 \ - \longrightarrow 3$$

- How can we use a stack to evaluate a postfix expression?
- 1 2 3 \* + 4 -

- How can we use a stack to evaluate a postfix expression?
- 1 2 3 \* + 4 -

When an operand is seen, ...

When an operator is seen, ...

When the expression is done, ...

- When an operand is seen, **push it onto the stack**

- When an operand is seen, **push it onto the stack**
- When an operator is seen, **the right and left operands are popped, the operation is evaluated, and the result is pushed back onto the stack**

- When an operand is seen, **push it onto the stack**
- When an operator is seen, **the right and left operands are popped, the operation is evaluated, and the result is pushed back onto the stack**
- When the expression is done, **the single item remaining on the stack is the answer**

1 2 3 \* + 4 -



1 2 3 \* + 4 -

↑  
operand  
push(1)



1 2 3 \* + 4 -

↑  
operand  
push(2)



1 2 3 \* + 4 -

↑  
operand  
push(3)



1 2 3 \* + 4 -



operator  
pop(), pop(), push(r)



1 2 3 \* + 4 -



operator  
pop(), pop(), push(r)

$$2 * 3 = 6$$



1 2 3 \* + 4 -



operator  
pop(), pop(), push(r)

$$2 * 3 = 6$$



1 2 3 \* + 4 -



operator  
pop(), pop(), push(r)



1 2 3 \* + 4 -

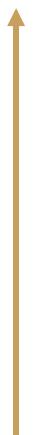


operator  
`pop()`, `pop()`, `push(r)`

$1 + 6 = 7$



1 2 3 \* + 4 -



operator  
pop(), pop(), push(r)

$$1 + 6 = 7$$



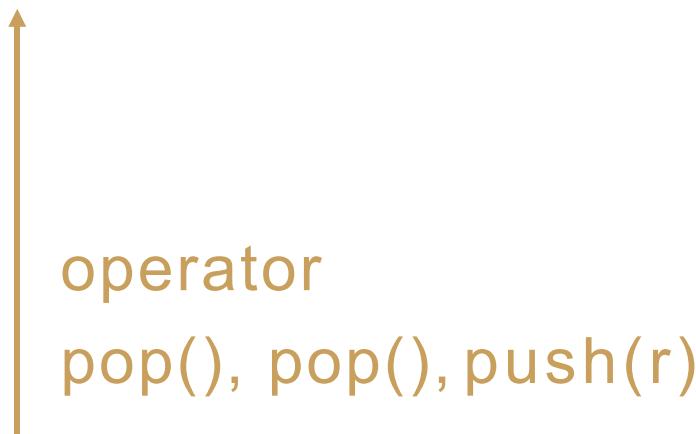
1 2 3 \* + 4 -



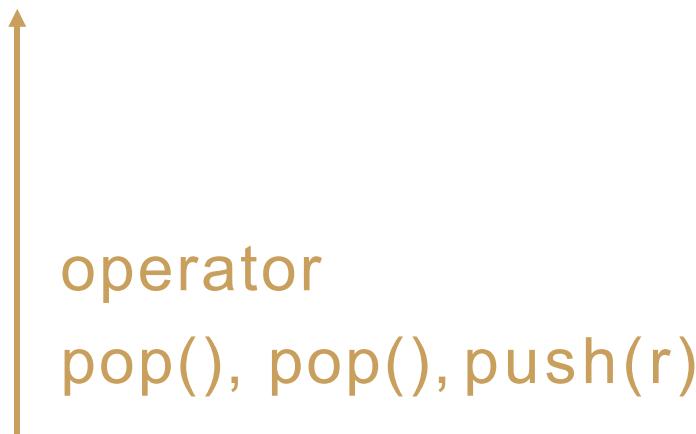
oper and  
push(4)



1 2 3 \* + 4 -

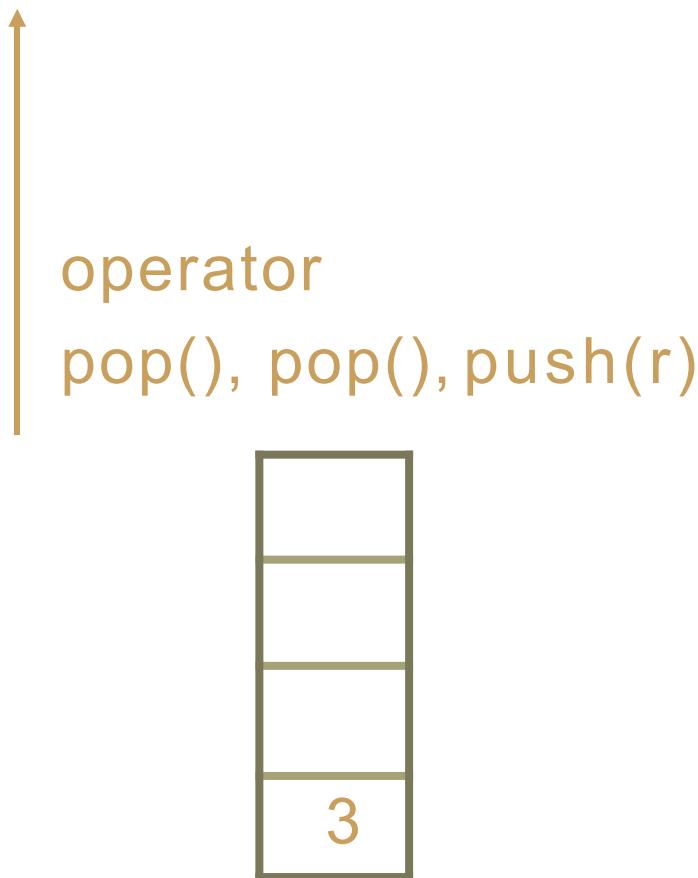


1 2 3 \* + 4 -



$$7 - 4 = 3$$

1 2 3 \* + 4 -



1 2 3 \* + 4 -

↑  
EOL  
pop()



1 2 3 \* + 4 -

↑  
EOL  
pop()



Answer is 3

```
1 // Java program to evaluate value of a postfix expression
2
3 import java.util.Stack;
4
5 public class Test
6 {
7     // Method to evaluate value of a postfix expression
8     static int evaluatePostfix(String exp)
9     {
10         //create a stack
11         Stack<Integer> stack=new Stack<>();
12
13         // Scan all characters one by one
14         for(int i=0;i<exp.length();i++)
15         {
16             char c=exp.charAt(i);
17
18             // If the scanned character is an opera
19             // push it to the stack.
20             if(Character.isDigit(c))
21                 stack.push(c - '0');
22
23             // If the scanned character is an operator,
24             // pop two elements from stack apply the operator
25             else
26             {
27                 int val1 = stack.pop();
28                 int val2 = stack.pop();
29
30                 switch(c)
31                 {
32                     case '+':
33                         stack.push(val2+val1);
34                         break;
35
36                     case '-':
37                         stack.push(val2- val1);
38                         break;
39
40                     case '/':
41                         stack.push(val2/val1);
42                         break;
43
44                     case '*':
45                         stack.push(val2*val1);
46                         break;
47
48                 }
49             }
50         }
51         return stack.pop();
52
53     // Driver program to test above functions
54     public static void main(String[] args)
55     {
56         String exp="231*+9-";
57         System.out.println("postfix evaluation: "+evaluatePostfix(exp));
58     }
59 }
```

# Call Stack (again!)

- Every time a method is invoked a unique *frame* is created
- When that method returns, execution resumes in the calling frame
- Methods return in reverse order in which they were called
  - FILO!
  - What method is the first in and last out?

```
1 // JAVA program to check whether two strings
2 // are anagrams of each other
3 import java.io.*;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 class GFG {
8
9     /* function to check whether two strings are
10    anagram of each other */
11    static boolean areAnagram(char[] str1, char[] str2)
12    {
13        // Get lengths of both strings
14        int n1 = str1.length;
15        int n2 = str2.length;
16
17        // If length of both strings is not same,
18        // then they cannot be anagram
19        if (n1 != n2)
20            return false;
21
22        // Sort both strings
23        Arrays.sort(str1);
24        Arrays.sort(str2);
25
26        // Compare sorted strings
27        for (int i = 0; i < n1; i++)
28            if (str1[i] != str2[i])
29                return false;
30
31        return true;
32    }
33
34    /* Driver program to test to print printDups*/
35    public static void main(String args[])
36    {
37        char str1[] = { 't', 'e', 's', 't' };
38        char str2[] = { 't', 't', 'e', 'w' };
39        if (areAnagram(str1, str2))
40            System.out.println("The two strings are"
41                               + " anagram of each other");
42        else
43            System.out.println("The two strings are not"
44                               + " anagram of each other");
45    }
46 }
```

↔ Arrays.equals(str1, str2);

The diagram illustrates a call stack with a single frame. It consists of two vertical brown lines representing the stack boundaries. Inside, there is a rectangular box with a light orange background and a dark orange border. The word "main" is centered within this box in a dark gray font.

main

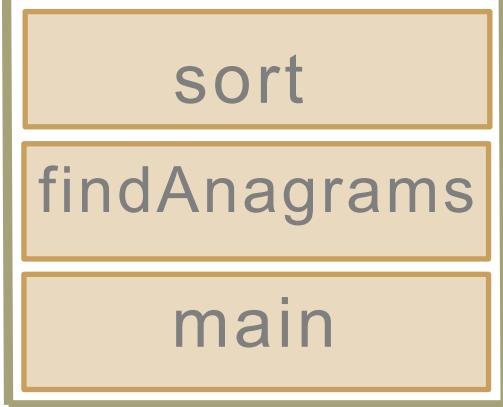
**call stack**

A call stack diagram consisting of two rectangular frames stacked vertically. The top frame is orange with a thin black border and contains the text "findAnagrams" in a dark gray sans-serif font. The bottom frame is a lighter shade of orange with a thin black border and contains the text "main" in a dark gray sans-serif font. Both frames are centered horizontally. To the left of the frames, there are two vertical brown lines extending upwards from the bottom line, creating a bracket-like structure. Below the frames, the word "call stack" is written in a large, bold, dark brown sans-serif font.

findAnagrams

main

**call stack**



**call stack**

A call stack diagram consisting of four rectangular frames stacked vertically. Each frame has a thin brown border and contains a method name in a dark gray sans-serif font. The frames are centered between two vertical brown lines. The methods listed from top to bottom are: compare, sort, findAnagrams, and main.

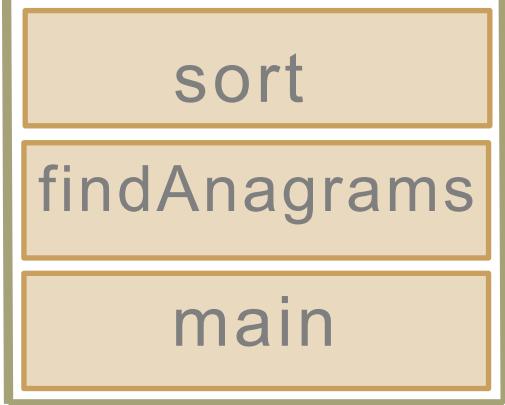
compare

sort

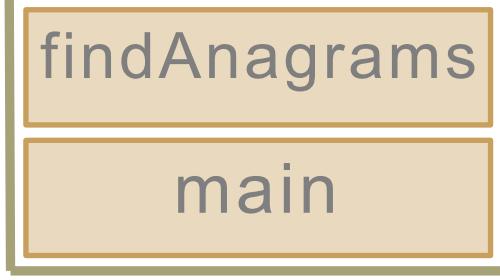
findAnagrams

main

**call stack**



**call stack**



findAnagrams

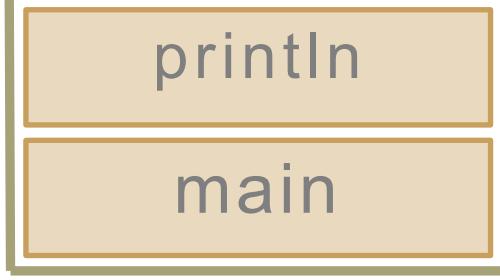
main

**call stack**

The diagram illustrates a call stack with a single frame. It consists of two vertical brown lines representing the stack boundaries. Inside, there is a rectangular box with a light orange background and a dark orange border. The word "main" is centered within this box in a dark gray font.

main

**call stack**



printIn

main

**call stack**

The diagram illustrates a call stack with a single frame. It consists of two vertical brown lines representing the stack boundaries. Inside, there is a rectangular box with a light orange background and a dark orange border. The word "main" is centered within this box in a dark gray font.

main

**call stack**

Next Time...

- Queues
  - Reading: Chapter 14
  - Quiz: stack, sorting
- Read the book chapter **before** and **after** session

Linked List

CSC220 | Computer Programming 2

Last Time...

# Stacks

- A **stack** is a FIRST-IN, LAST-OUT data structure
  - FILO
- Operations:
  - *push*... adds an item **on to the top** of the stack
  - *pop*... removes and returns the item **on the top** of the stack

# Queues

- A **queue** is a FIRST-IN, FIRST-OUT data structure
  - **FIFO**
- Operations:
  - *enqueue*... adds an item to the **back** of the queue
  - *dequeue*... removes and returns the item at the **front**

Today...

# Testing

# What does it take to write a program

- You first need to get the logic right!
  - Use pen and paper
  - Write down the algorithm in English

# What does it take to write a program

- You first need to get the logic right!
  - Use pen and paper
  - Write down the algorithm in English
- Then, you need to convert the logic into code
  - Be careful about Syntax
  - You need CSC120 concept at your finger tips!

# What does it take to write a program

- You first need to get the logic right!
  - Use pen and paper
  - Write down the algorithm in English
- Then, you need to convert the logic into code
  - Be careful about Syntax
  - You need CSC120 concept at your finger tips!
- Debugging/testing!

# Testing

- This is a skill 😊
- Assuming your code works (compiles/run)
  - If not you have a problem in the logic or syntax or both!
- Okay, now how do I do testing?

# Testing

- Testing should be a **sequential** process
  1. Test each unit!
  2. Start from basics!
  3. Only after the other two think about “special cases”

# Testing

- Testing should be a **sequential** process
  1. Test each unit!
  2. Start from basics!
  3. Only after the other two think about “special cases”
- Example:
  - Your library assignment
    - Can I test “checkin” before “checkout”?

# Testing a method

- What does a method have?
  - Input
  - Logic
  - Output

# Testing a method

- What does a method have?
  - Input
  - Logic
  - Output
- Is the method receiving all the input it needs?

# Testing a method

- What does a method have?
  - Input
  - Logic
  - Output
- What would happen if the method is called on “wrong” input?
- What if the input is null?
- ...

# Testing a method

- What does a method have?
  - Input
  - Logic
  - Output
- Given the possibilities for the input, does the logic of my method work for all of them?

# Testing a method

- What does a method have?
  - Input
  - Logic
  - Output
- Is my method supposed to return something?
- If so, I am constructing the output properly?

# Linked List

# Memory primer

- All data in your program resides in memory at some point during its life
- Think of memory as giant blocks of bytes:
- Each byte has its own memory address (a number)
- Memory is ordered

# Memory in Java

- What actually happens when you use the new keyword?

- What actually happens when you use the `new` keyword?
- `new` instructs the system to find a contiguous block of bytes big enough to hold whatever you are creating

- What actually happens when you use the `new` keyword?
- `new` instructs the system to find a contiguous block of bytes big enough to hold whatever you are creating
  - `int arr[] = new int[10];`
  - *finds a block of memory big enough to hold 10 ints*

- What actually happens when you use the `new` keyword?
- `new` instructs the system to find a contiguous block of bytes big enough to hold whatever you are creating
  - `int arr[] = new int[10];`
    - *finds a block of memory big enough to hold 10 ints*
  - `arr[0]` is right next to `arr[1]` in memory
    - *the addresses of these two numbers are contiguous!*

- Arrays are a **random access** data structure
  - Any item in the array can be accessed instantly

- Arrays are a **random access** data structure
  - Any item in the array can be accessed instantly
- EXAMPLE
  - To access item **23** in an array of ints
    - Simply take the address of the beginning of the array
    - Add 23 times the size of each item
    - Address of **arr[23]** is address of **arr[0] + (23 \* 4)**

- Arrays are a **random access** data structure
  - Any item in the array can be accessed instantly
- No matter the size of the array, accessing item  $i$  can be done in  $O(c)$

- Arrays are a **random access** data structure
  - Any item in the array can be accessed instantly
- No matter the size of the array, accessing item  $i$  can be done in  $O(c)$ 
  - i.e., one addition and one multiplication

- each time you call new, the allocated block can be anywhere in memory

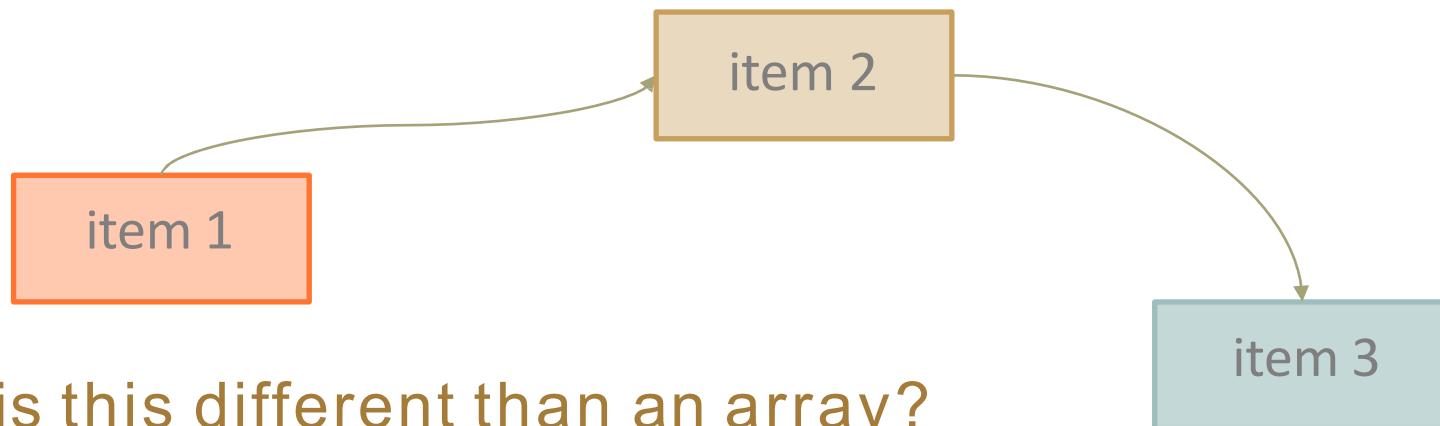
```
Circle c1 = new Circle();  
Circle c2 = new Circle();
```

- c1 **may** be at location 2048, and c2 **may** be at location 640
- you have no control over this!

# Linked Structures

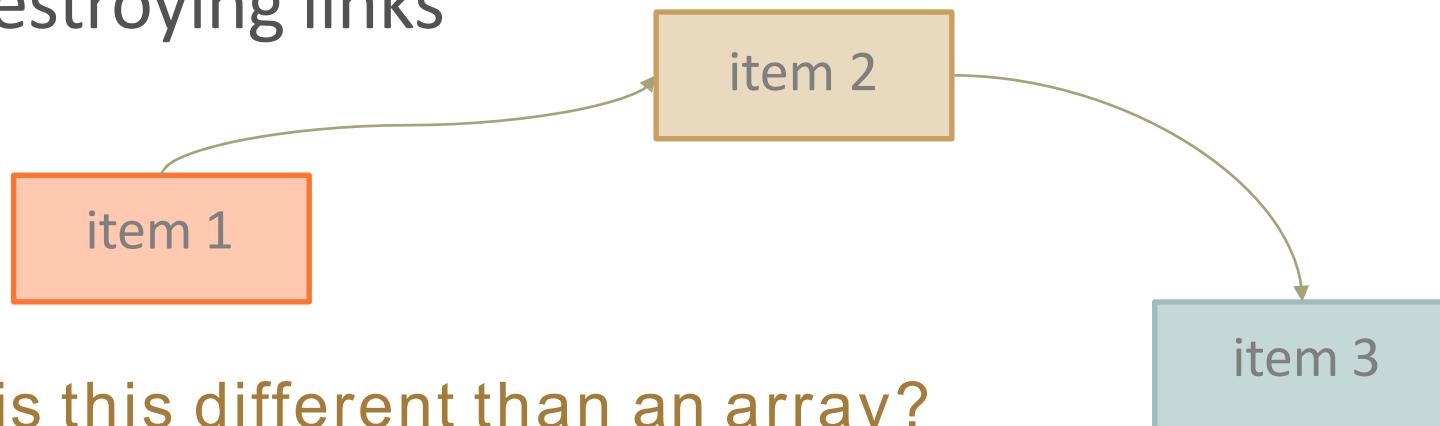
- **Linked structures**

- Data storage structure
- Individual items have *links* (references) to other items



How is this different than an array?

- **Linked structures**
  - Data storage structure
  - Individual items have *links* (references) to other items
- Items don't reside in a single contiguous block of memory
- Items can be *dynamically* added or removed from the structure, simply by creating or destroying links



How is this different than an array?

# Linked structures versus arrays

- Arrays provide **random access**
- Linked structures provide **sequential access**
- Insert/remove is not easy for arrays
- Linked structures are more flexible for insert/remove

- Linked structures have a reference to another instance of the structure
  - How?

- Linked structures have a reference to another instance of the structure
  - How?
  - Using an object called a **node**

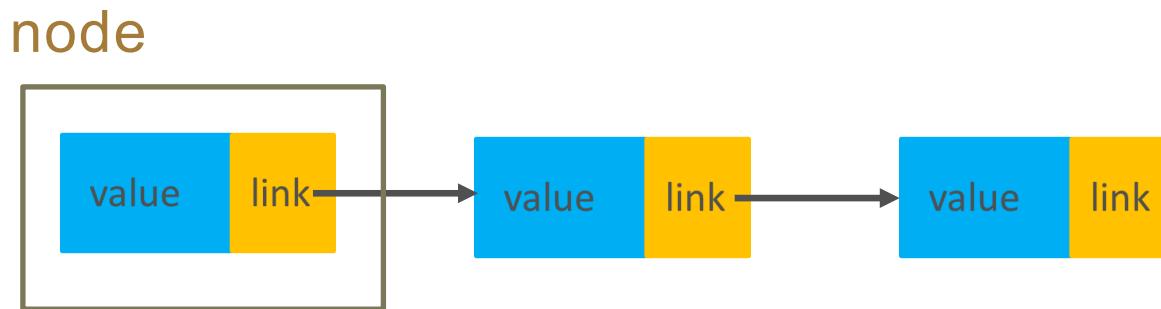
```
class LinkedNode {  
    int ID; //each node stores some data  
    LinkedNode next; //and one of itself!  
}
```

looks a bit like a **recursive** class definition

# Linked list

- A *linked list* is another way to implement a list
- Each **node**, or item in the list, has a link to the next item in the list

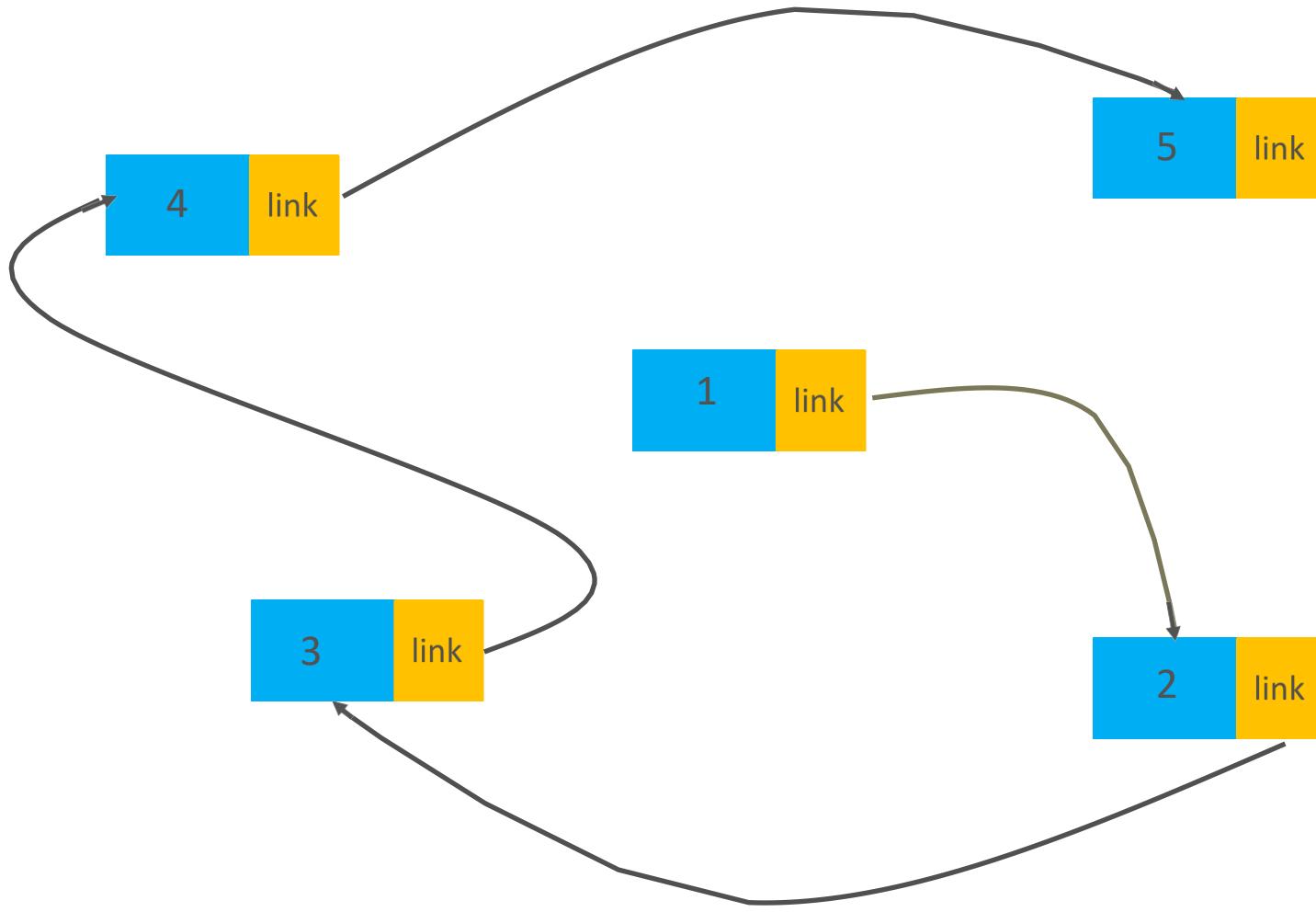
List is an interface in Java, no concrete implementation



- A single node consists of some **data** and a **reference** to another node

LinkedList<E>

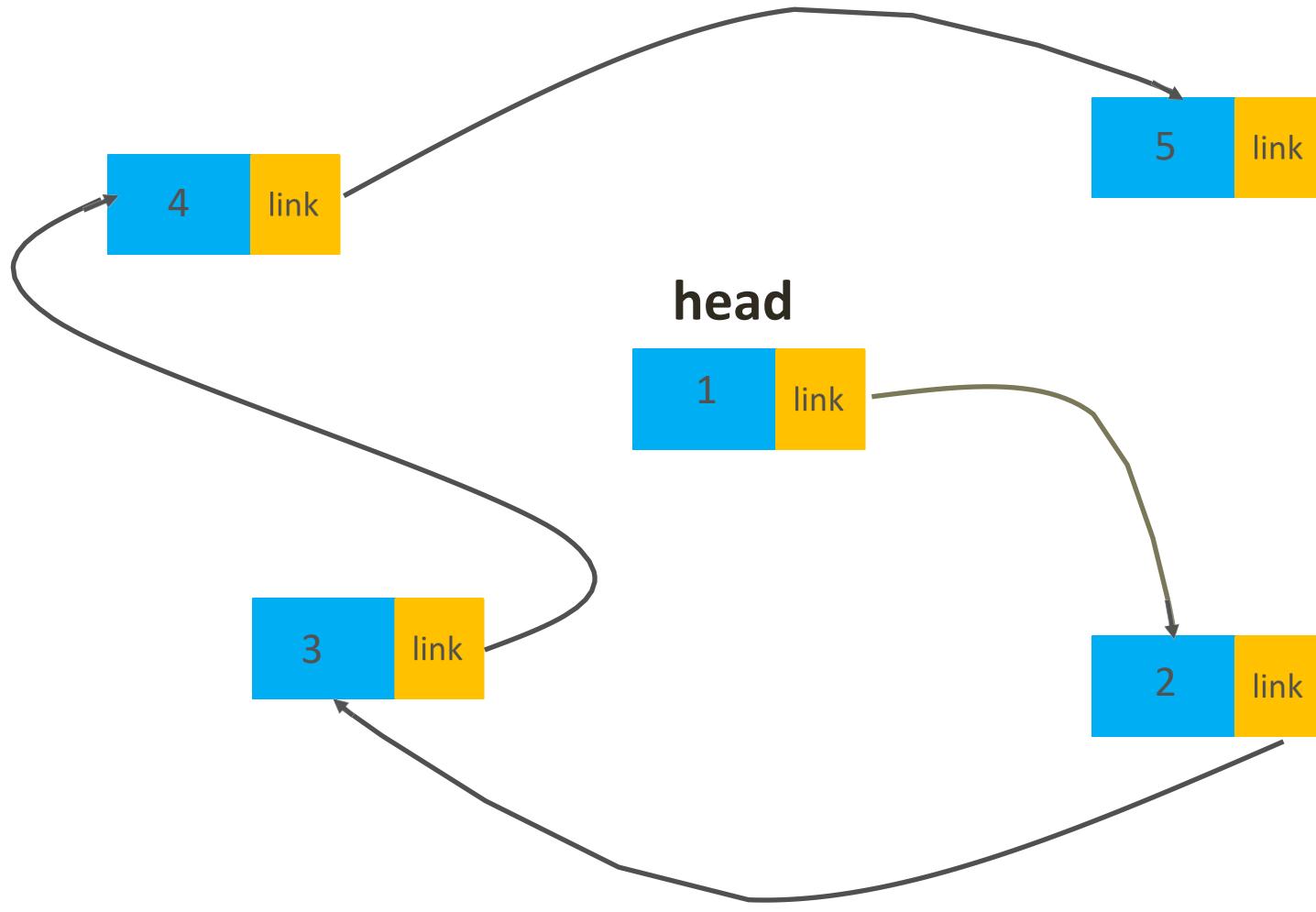
- Nodes may not be contiguous in memory!

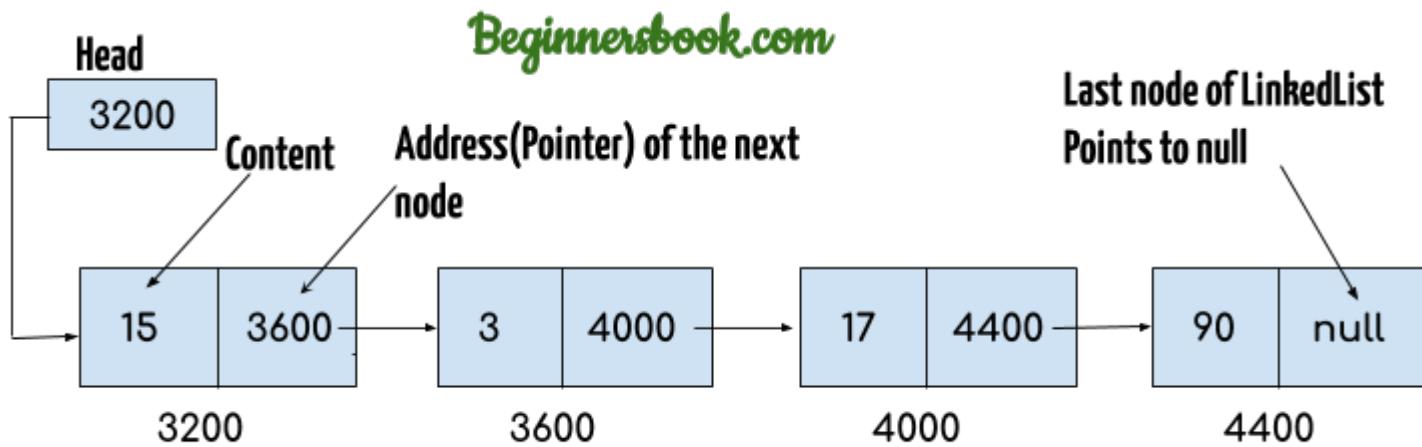


- With an array, we have a single variable that can access any item with [ ]
- With a linked list, how do we access individual elements?
  - HINT: we need somewhere to start

- With an array, we have a single variable that can access any item with [ ]
- With a linked list, how do we access individual elements?
  - HINT: we need somewhere to start
- Always keep track of the first node
  - called the **head**

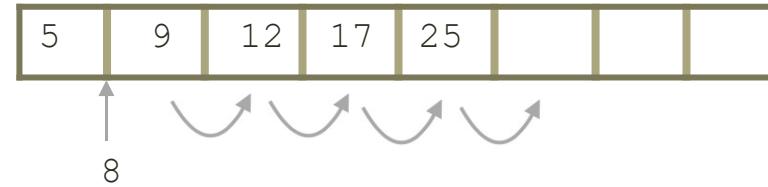
- With an array, we have a single variable that can access any item with [ ]
- With a linked list, how do we access individual elements?
  - HINT: we need somewhere to start
- Always keep track of the first node
  - called the **head**
- From the head node we can access any other node by following the links





# Insertion and Deletion

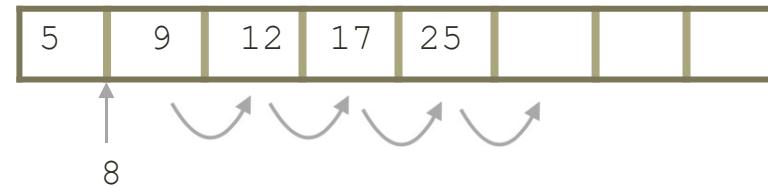
- Insertion into an array:



- Insert into an array list

- Insertion into an array:

What is the complexity?

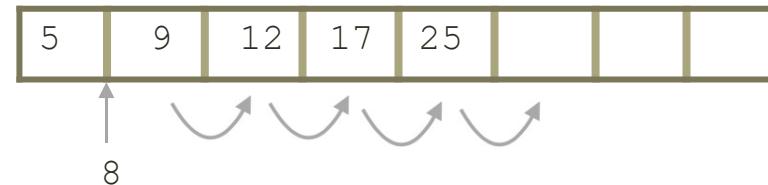


1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$

- Insert into an array list

- Insertion into an array:

What is the complexity?



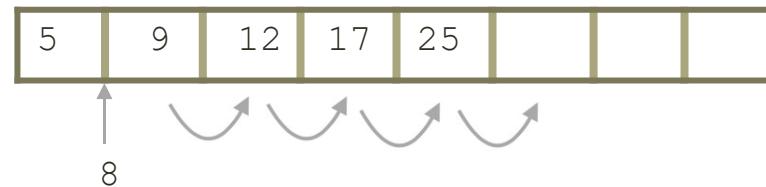
1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$

- Insert into an array list



- Insertion into an array:

What is the complexity?



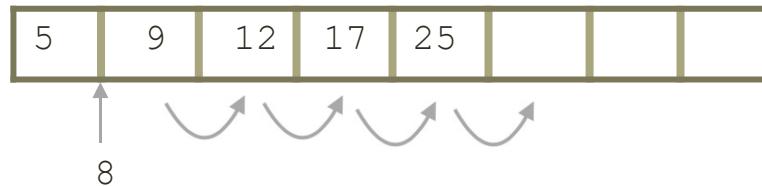
1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$

- Insert into an array list



- Insertion into an array:

What is the complexity?

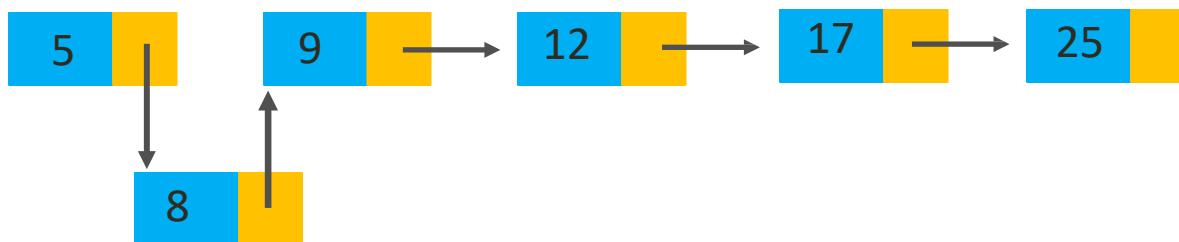


1.  $c$
2.  $\log N$
3.  $N$
4.  $N \log N$
5.  $N^2$
6.  $N^3$

- Insert into an array list



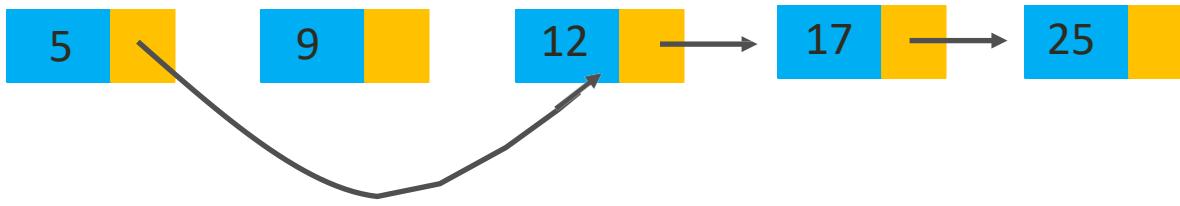
8



# Deletion from a linkedlist



# Deletion from a linkedlist



# Deletion from a linkedlist

- Deletion from a linked list



9 is now stranded  
garbage collector will clean it up

# Linked list vs arrays

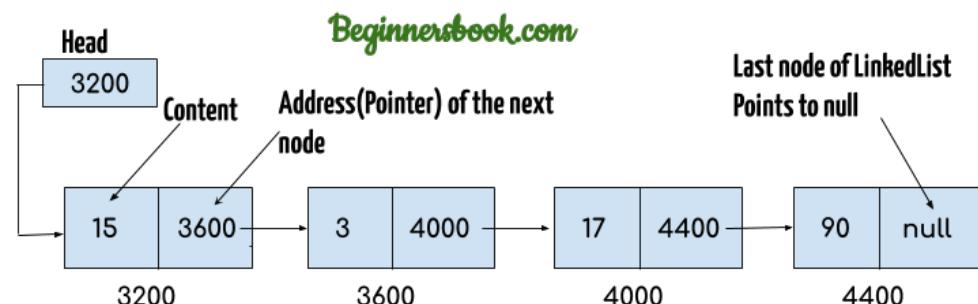
- Cost of accessing a random item at location  $i$ ?  
1.  $c$   
2.  $\log N$   
3.  $N$   
4.  $N \log N$   
5.  $N^2$   
6.  $N^3$
- Cost of removeFirst()?
- Cost of addFirst()?

# Implementation Details

```

1 import java.io.*;
2
3 // Java program to implement
4 // a Singly Linked List
5 public class LinkedList {
6
7     Node head; // head of list
8
9     // Linked list Node.
10    // This inner class is made static
11    // so that main() can access it
12    static class Node {
13
14        int data;
15        Node next;
16
17        // Constructor
18        Node(int d)
19        {
20            data = d;
21            next = null;
22        }
23    }
24
25    // *****INSERTION*****
26
27    // Method to insert a new node
28    public static LinkedList insert(LinkedList list, int data)
29    {
30        // Create a new node with given data
31        Node new_node = new Node(data);
32        new_node.next = null;
33
34        // If the Linked List is empty,
35        // then make the new node as head
36        if (list.head == null) {
37            list.head = new_node;
38        }
39        else {
40            // Else traverse till the last node
41            // and insert the new_node there
42            Node last = list.head;
43            while (last.next != null) {
44                last = last.next;
45            }
46
47            // Insert the new_node at last node
48            last.next = new_node;
49        }
50
51        // Return the list by head
52        return list;
53    }
54}

```



```

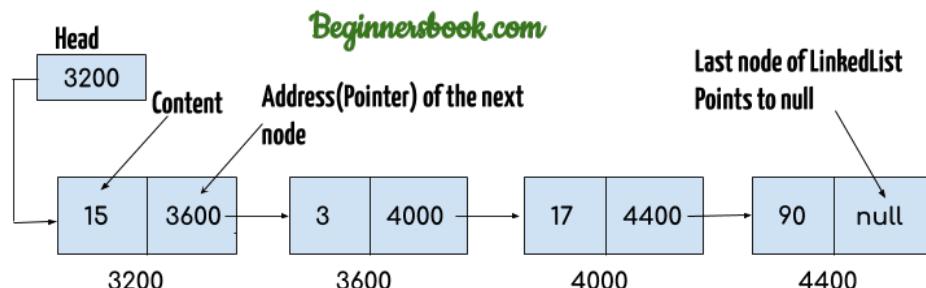
55 // ****TRAVERSAL*****
56
57 // Method to print the LinkedList.
58 public static void printList(LinkedList list)
59 {
60     Node currNode = list.head;
61
62     System.out.print("\nLinkedList: ");
63
64     // Traverse through the LinkedList
65     while (currNode != null) {
66         // Print the data at current node
67         System.out.print(currNode.data + " ");
68
69         // Go to next node
70         currNode = currNode.next;
71     }
72     System.out.println("\n");
73 }
74

```

```

75 // *****DELETION BY KEY*****
76
77 // Method to delete a node in the LinkedList by KEY
78 public static LinkedList deleteByKey(LinkedList list, int key)
79 {
80     // Store head node
81     Node currNode = list.head, prev = null;
82
83     //
84     // CASE 1:
85     // If head node itself holds the key to be deleted
86
87     if (currNode != null && currNode.data == key) {
88         list.head = currNode.next; // Changed head
89
90         // Display the message
91         System.out.println(key + " found and deleted");
92
93         // Return the updated List
94         return list;
95     }

```

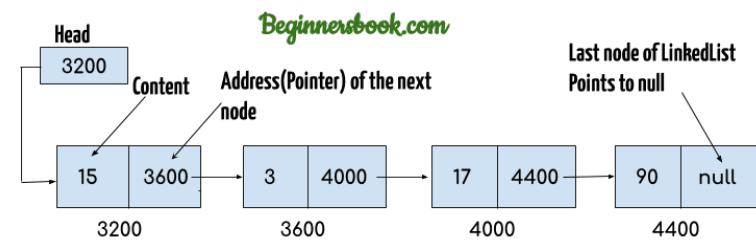


Last node of LinkedList  
Points to null

```

96
97
98 // CASE 2:
99 // If the key is somewhere other than at head
100 //
101
102 // Search for the key to be deleted,
103 // keep track of the previous node
104 // as it is needed to change currNode.next
105 while (currNode != null && currNode.data != key) {
106     // If currNode does not hold key
107     // continue to next node
108     prev = currNode;
109     currNode = currNode.next;
110 }
111
112 // If the key was present, it should be at currNode
113 // Therefore the currNode shall not be null
114 if (currNode != null) {
115     // Since the key is at currNode
116     // Unlink currNode from linked list
117     prev.next = currNode.next;
118
119     // Display the message
120     System.out.println(key + " found and deleted");
121 }
122
123 //
124 // CASE 3: The key is not present
125 //
126
127 // If key was not present in linked list
128 // currNode should be null
129 if (currNode == null) {
130     // Display the message
131     System.out.println(key + " not found");
132 }
133
134 // return the List
135 return list;
136 }

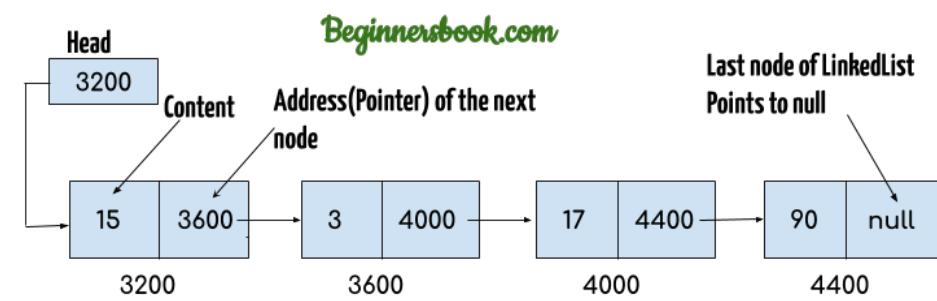
```



```

138 // ****DELETION AT A POSITION*****
139
140 // Method to delete a node in the LinkedList by POSITION
141 public static LinkedList deleteAtPosition(LinkedList list, int index)
142 {
143     // Store head node
144     Node currNode = list.head, prev = null;
145
146     //
147     // CASE 1:
148     // If index is 0, then head node itself is to be deleted
149
150     if (index == 0 && currNode != null) {
151         list.head = currNode.next; // Changed head
152
153         // Display the message
154         System.out.println(index + " position element deleted");
155
156         // Return the updated List
157         return list;
158     }
159
160     //
161     // CASE 2:
162     // If the index is greater than 0 but less than the size of Linked
163     //
164     // The counter
165     int counter = 0;
166
167     // Count for the index to be deleted,
168     // keep track of the previous node
169     // as it is needed to change currNode.next
170     while (currNode != null) {
171
172         if (counter == index) {
173             // Since the currNode is the required position
174             // Unlink currNode from linked list
175             prev.next = currNode.next;
176
177             // Display the message
178             System.out.println(index + " position element deleted");
179             break;
180         }
181         else {
182             // If current position is not the index
183             // continue to next node
184             prev = currNode;
185             currNode = currNode.next;
186             counter++;
187         }
188     }
189 }

```



```
189
190     // If the position element was found, it should be at currNode
191     // Therefore the currNode shall not be null
192     //
193     // CASE 3: The index is greater than the size of the LinkedList
194     //
195     // In this case, the currNode should be null
196     if (currNode == null) {
197         // Display the message
198         System.out.println(index + " position element not found");
199     }
200
201     // return the List
202     return list;
203 }
```

```
205 // *****MAIN METHOD*****
206
207 // method to create a Singly linked list with n nodes
208 public static void main(String[] args)
209 {
210     /* Start with the empty list. */
211     LinkedList list = new LinkedList();
212
213     //
214     // *****INSERTION*****
215     //
216
217     // Insert the values
218     list = insert(list, 1);
219     list = insert(list, 2);
220     list = insert(list, 3);
221     list = insert(list, 4);
222     list = insert(list, 5);
223     list = insert(list, 6);
224     list = insert(list, 7);
225     list = insert(list, 8);
226
227     // Print the LinkedList
228     printList(list);
229
230     //
231     // *****DELETION BY KEY*****
232     //
233
234     // Delete node with value 1
235     // In this case the key is ***at head***
236     deleteByKey(list, 1);
237
238     // Print the LinkedList
239     printList(list);
240
241     // Delete node with value 4
242     // In this case the key is present ***in the middle***
243     deleteByKey(list, 4);
244
245     // Print the LinkedList
246     printList(list);
247
248     // Delete node with value 10
249     // In this case the key is ***not present***
250     deleteByKey(list, 10);
251
252     // Print the LinkedList
253     printList(list);
```

```
254
255    //
256    // *****DELETION AT POSITION*****
257    //
258
259    // Delete node at position 0
260    // In this case the key is ***at head***
261    deleteAtPosition(list, 0);
262
263    // Print the LinkedList
264    printList(list);
265
266    // Delete node at position 2
267    // In this case the key is present ***in the middle***
268    deleteAtPosition(list, 2);
269
270    // Print the LinkedList
271    printList(list);
272
273    // Delete node at position 10
274    // In this case the key is ***not present***
275    deleteAtPosition(list, 10);
276
277    // Print the LinkedList
278    printList(list);
279 }
280 }
```

How about using generic programming or  
using Java's implementation?

```
1 import java.util.Currency;
2
3 /**
4 * LinkList implementation with generics
5 *
6 * @author malalanayake
7 *
8 * @param <T>
9 */
10 public class LinkedListWithGenerics<T> {
11     private LinkedNode<T> first;
12     private LinkedNode<T> last;
13     private int count;
14
15     /**
16      * Internal linked node implementation
17      *
18      * @author malalanayake
19      *
20      * @param <T>
21      */
22     private class LinkedNode<T> {
23         private T data;
24         private LinkedNode<T> next;
25
26         public LinkedNode() {
27             this.data = null;
28             this.next = null;
29         }
30     }
```

```
30
31     public LinkedNode(T obj) {
32         this.data = obj;
33         this.next = null;
34     }
35
36     public T getData() {
37         return data;
38     }
39
40     public void setData(T data) {
41         this.data = data;
42     }
43
44     public LinkedNode<T> getNext() {
45         return next;
46     }
47
48     public void setNext(LinkedNode<T> next) {
49         this.next = next;
50     }
51
52 }
53
54 public LinkedListWithGenerics() {
55     LinkedNode<T> newLiked = new LinkedNode<T>();
56     this.first = newLiked;
57     this.last = this.first;
58 }
59 }
```

```
60         /**
61          * Add values to the list
62          *
63          * @param data
64          */
65         public void add(T data) {
66             LinkedNode<T> newData = new LinkedNode<T>(data);
67             if (this.first.getData() == null) {
68                 this.first = newData;
69                 this.last = this.first;
70             } else {
71                 this.last.setNext(newData);
72                 this.last = newData;
73             }
74             count++;
75         }
76     }
```

```
77     /**
78      * Remove values from the list
79      *
80      * @param data
81      */
82     public void remove(T data) {
83         LinkedNode<T> current = first;
84         if (this.first.getData().equals(data)) {
85             if (this.first.getNext() == null) {
86                 LinkedNode<T> newNode = new LinkedNode<T>();
87                 this.first.setData(null);
88                 this.first = newNode;
89                 this.last = this.first;
90             } else {
91                 this.first.setData(null);
92                 this.first = this.first.getNext();
93             }
94         } else {
95             boolean wasDeleted = false;
96             while (!wasDeleted) {
97                 LinkedNode<T> currentNext = current.getNext();
98                 if (currentNext.getData().equals(data)) {
99                     currentNext.setData(null);
100                    current.setNext(currentNext.getNext());
101                    currentNext = null;
102                    wasDeleted = true;
103                    count--;
104                } else {
105                    current = current.getNext();
106                }
107            }
108        }
109    }
```

```
111     public void print() {
112         boolean allPrinted = false;
113         LinkedNode<T> crr = first;
114         System.out.print("[");
115         while (!allPrinted) {
116             if (crr.getData() != null) {
117                 if (crr.getNext() != null) {
118                     System.out.print(crr.getData().toString() + ",");
119                     LinkedNode<T> crrNext = crr.getNext();
120                     crr = crrNext;
121                 } else {
122                     System.out.print(crr.getData().toString() + "]");
123                     allPrinted = true;
124                 }
125             } else {
126                 allPrinted = true;
127             }
128         }
129         System.out.println();
130     }
131
132     public int getCount() {
133         return count;
134     }
```

```
136     public static void main(String[] args) {  
137         LinkedListWithGenerics<String> linkedLst = new LinkedListWithGenerics<String>();  
138         linkedLst.add("Test");  
139         linkedLst.add("Free");  
140         linkedLst.add("Yes");  
141         linkedLst.add("Me");  
142  
143         linkedLst.print();  
144         System.out.println(linkedLst.getCount());  
145  
146         linkedLst.remove("Me");  
147  
148         linkedLst.print();  
149  
150         System.out.println(linkedLst.getCount());  
151  
152     }  
153 }
```

Even simpler: use Java's implementation

```
1 // Java code for Linked List implementation
2
3 import java.util.*;
4
5 public class Test
6 {
7     public static void main(String args[])
8     {
9         // Creating object of class linked list
10        LinkedList<String> object = new LinkedList<String>();
11
12        // Adding elements to the linked list
13        object.add("A");
14        object.add("B");
15        object.addLast("C");
16        object.addFirst("D");
17        object.add(2, "E");
18        object.add("F");
19        object.add("G");
20        System.out.println("Linked list : " + object);
21
22        // Removing elements from the linked list
23        object.remove("B");
24        object.remove(3);
25        object.removeFirst();
26        object.removeLast();
27        System.out.println("Linked list after deletion: " + object);
28
29        // Finding elements in the linked list
30        boolean status = object.contains("E");
31
32        if(status)
33            System.out.println("List contains the element 'E' ");
34        else
35            System.out.println("List doesn't contain the element 'E'");
36
37        // Number of elements in the linked list
38        int size = object.size();
39        System.out.println("Size of linked list = " + size);
40
41        // Get and set elements from linked list
42        Object element = object.get(2);
43        System.out.println("Element returned by get() : " + element);
44        object.set(2, "Y");
45        System.out.println("Linked list after change : " + object);
46    }
47 }
```

## [java.util.LinkedList](#)

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

# LinkedList vs ArrayList

## LinkedList vs ArrayList

|                                                      |        |        |
|------------------------------------------------------|--------|--------|
| Insertion & deletion<br>(assuming position is known) | $O(c)$ | $O(N)$ |
| Accessing a random item                              | $O(N)$ | $O(c)$ |

- Choose the structure based on the expected use
  - what is the common case?
- What if insertion / deletion is always from the front / end?

A **dynamic array** is an [array](#) with a big improvement: automatic resizing.

| ArrayList                                                                                                                                                                                 | LinkedList                                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1) ArrayList internally uses a <b>dynamic array</b> to store the elements.                                                                                                                | LinkedList internally uses a <b>doubly linked list</b> to store the elements.                                                                                         |
| 2) Manipulation (insertion and deletion) with ArrayList is <b>slow</b> because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation (insertion and deletion) with LinkedList is <b>faster</b> than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can <b>act as a list</b> only because it implements List only.                                                                                                      | LinkedList class can <b>act as a list and queue</b> both because it implements List and Deque interfaces.                                                             |
| 4) ArrayList is <b>better for storing and accessing</b> data.                                                                                                                             | LinkedList is <b>better for manipulating</b> data.                                                                                                                    |

# LinkedList vs array

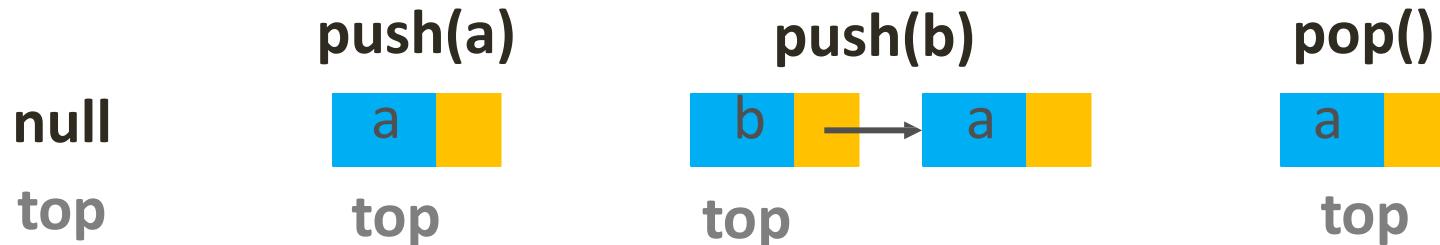
- Cost of accessing a random item at location  $i$ ?
- Cost of `removeFirst()`?
- Cost of `addFirst()`?
  - 1.  $c$
  - 2.  $\log N$
  - 3.  $N$
  - 4.  $N \log N$
  - 5.  $N^2$
  - 6.  $N^3$

# Stack as a linkedlist...

- Treat the **head** as the **top** of the stack
- To push, add to the beginning of the linked list
- To pop, return the top and remove the first item

# Stack as a linkedlist...

- Treat the **head** as the **top** of the stack
- To push, add to the beginning of the linked list
- To pop, return the top and remove the first item

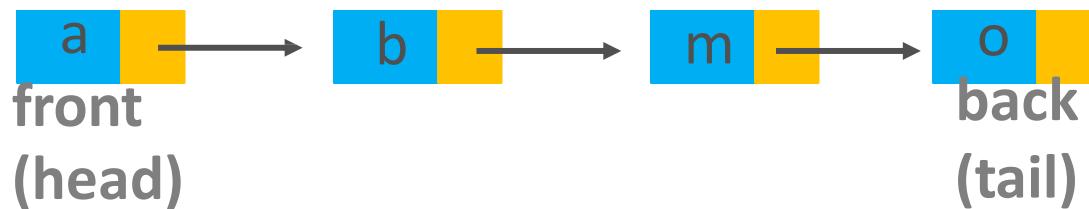


# Performance

- Linked lists never incur the penalty of resizing
  - Adds to a linked list are always  $O(1)$
- No wasted extra array space
  - All stack operations are  $O(1)$
- A stack can be easily implemented on top of an existing linked list with very little extra code!

# Queue as a linkedlist...

- Remember, inserting and deleting to the head and tail of a linked list is automatically  $O(1)$
- `front` is analogous to `head`
- `back` is analogous to `tail`
- No messy wrap-around, or growth issues



- Which linked list operations are analogous to *enqueue* and *dequeue*?

# Queue as a linkedlist...

```
Queue<Integer> q = new Queue<Integer>();
```

A thin grey horizontal arrow pointing to the right, positioned above the word "error".

error

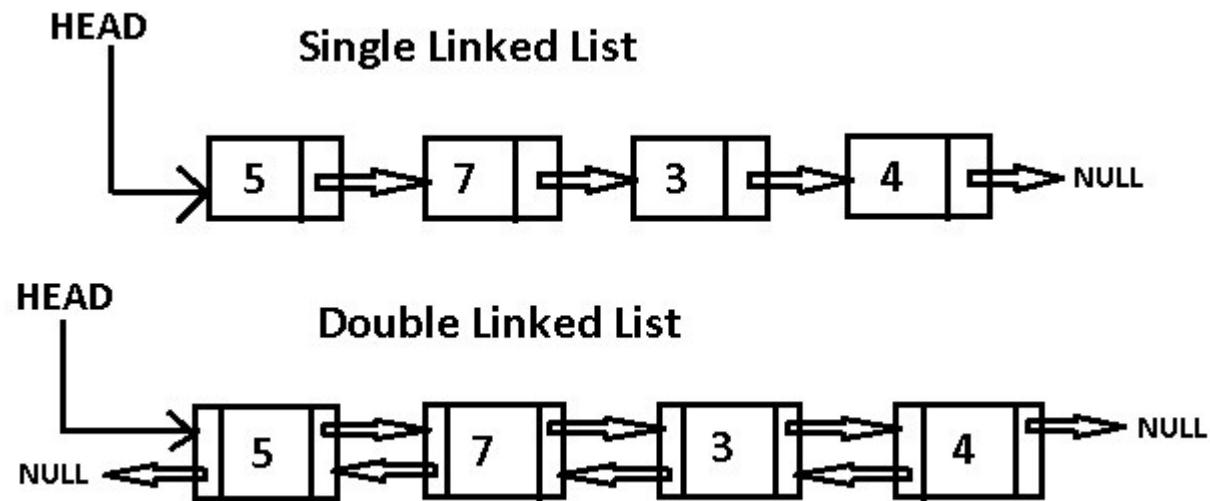
Queue is an interface in Java

```
Queue<Integer> q = new LinkedList<Integer>();  
q.add(3); // enqueue  
q.add(5); // enqueue  
q.remove(); // dequeue
```

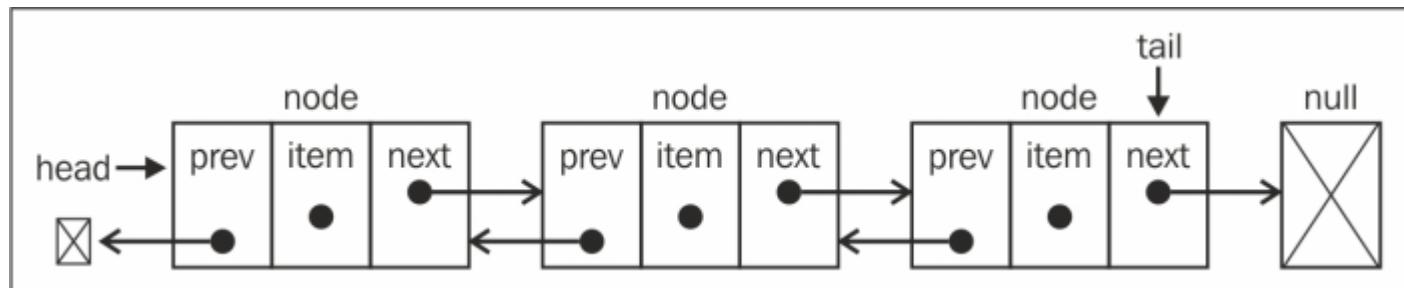
# Summary

- Linked lists and wrap-around arrays are both  $O(1)$  for queue implementations
- BUT, arrays are much more complicated to code
- **Both queues and stacks require very little code on top of a good linked list implementation**

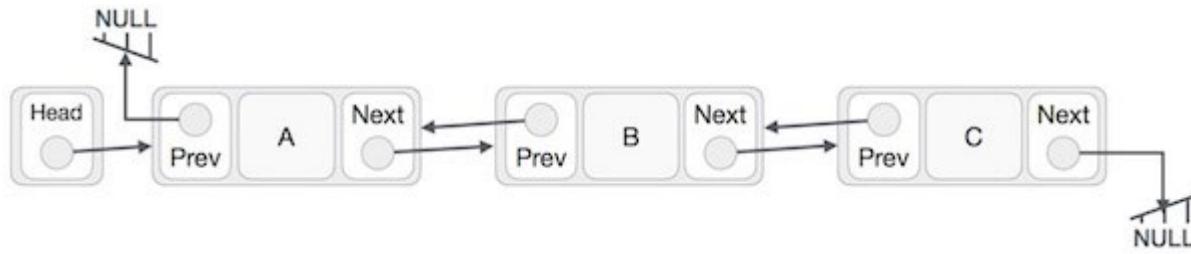
# Doubly-linked list



- Nodes have a link to `next` *and* `previous` node
- Allows for traversal in either forward or reverse order
- Maintains a `tail` node as well as a `head` node

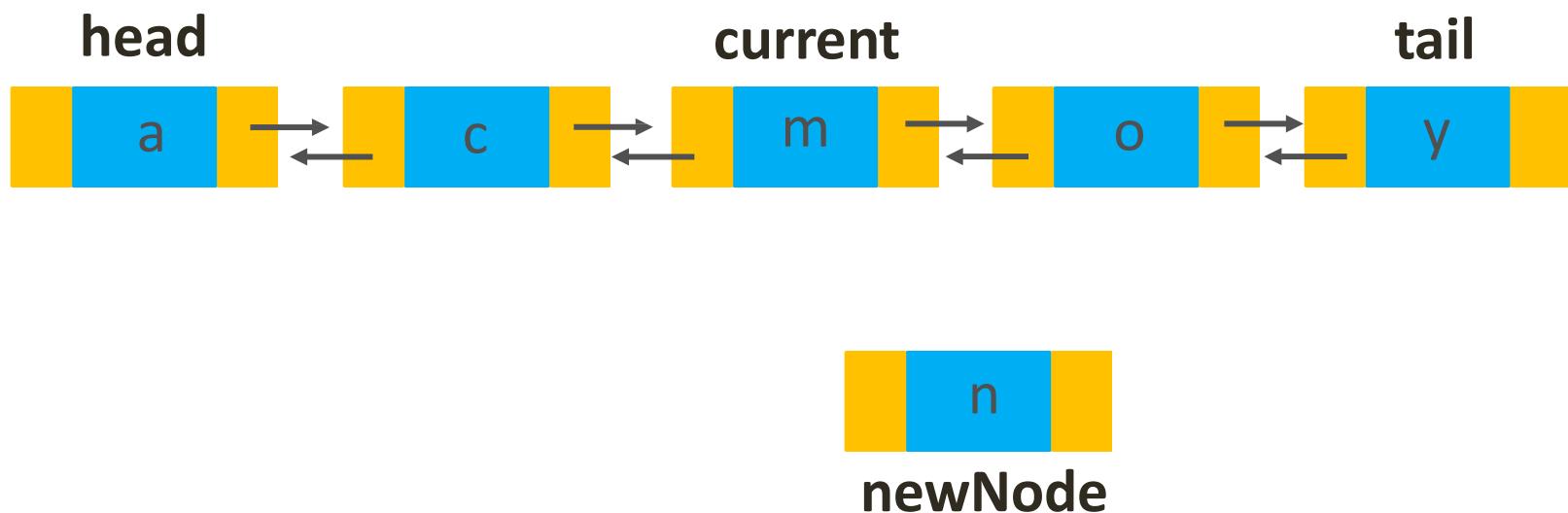


# Doubly-linked list insertion



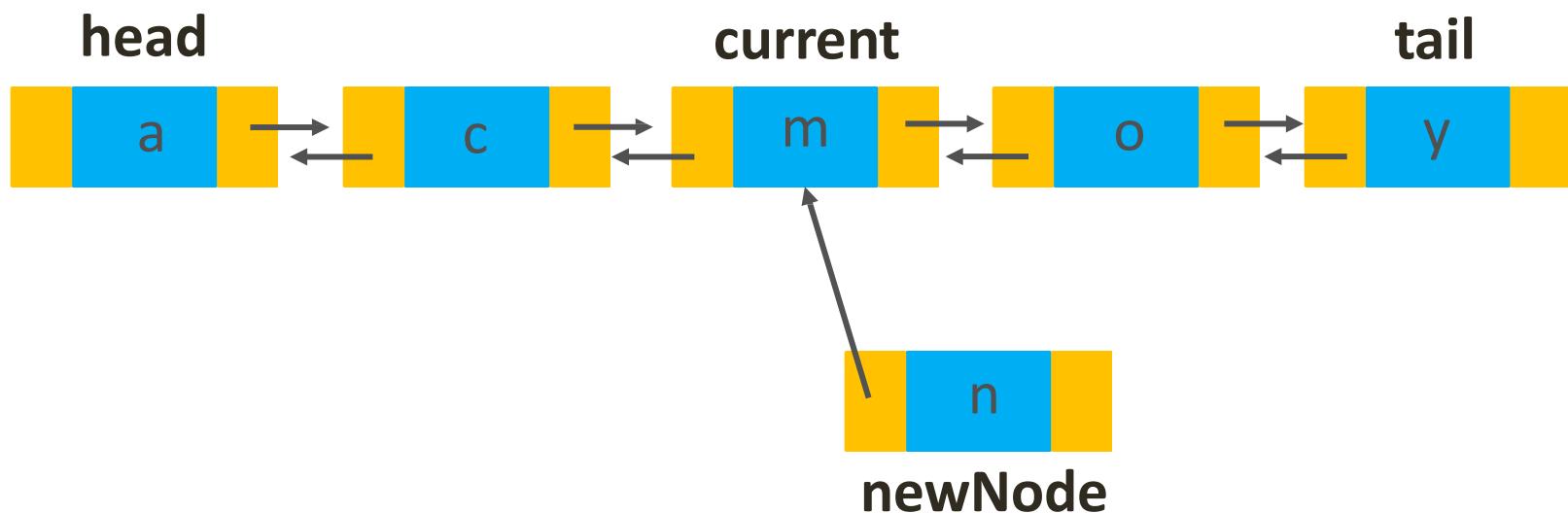
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';
```



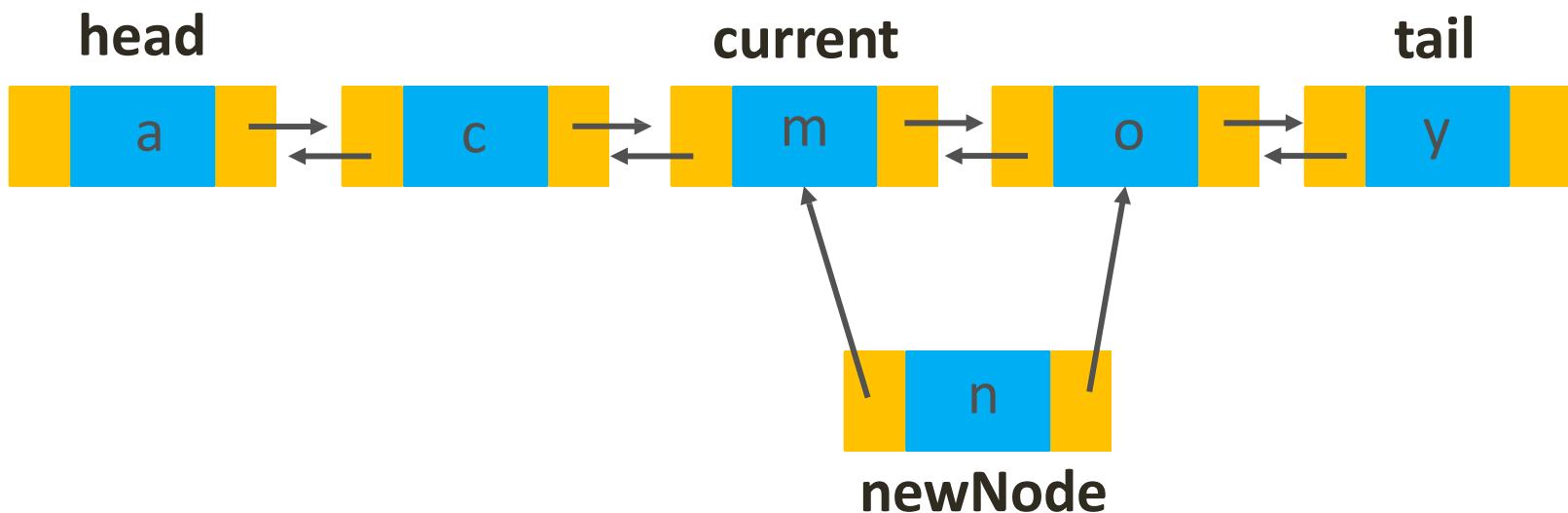
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;
```



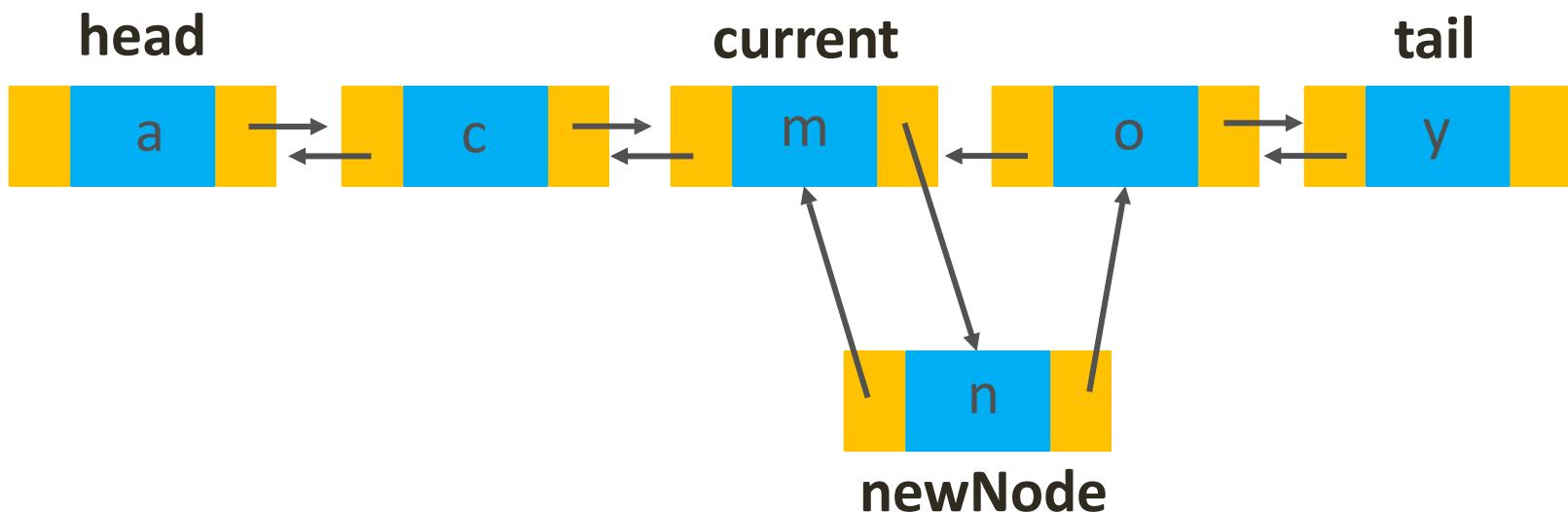
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;
```



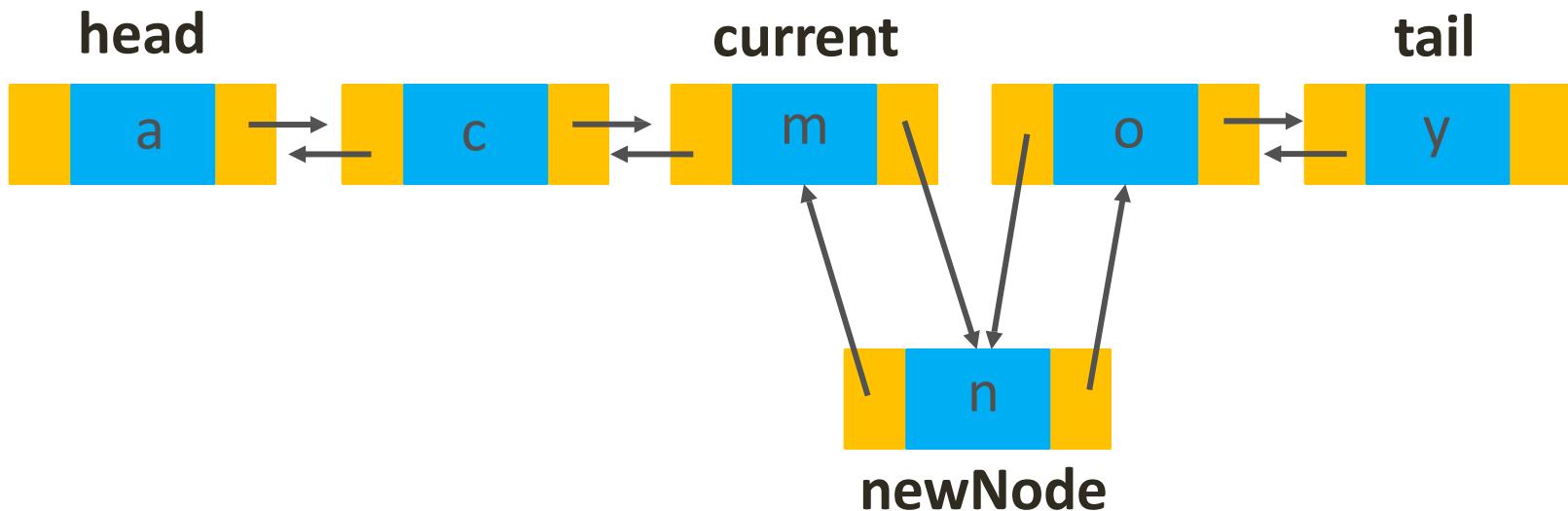
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;
```



# Doubly-linked list insertion

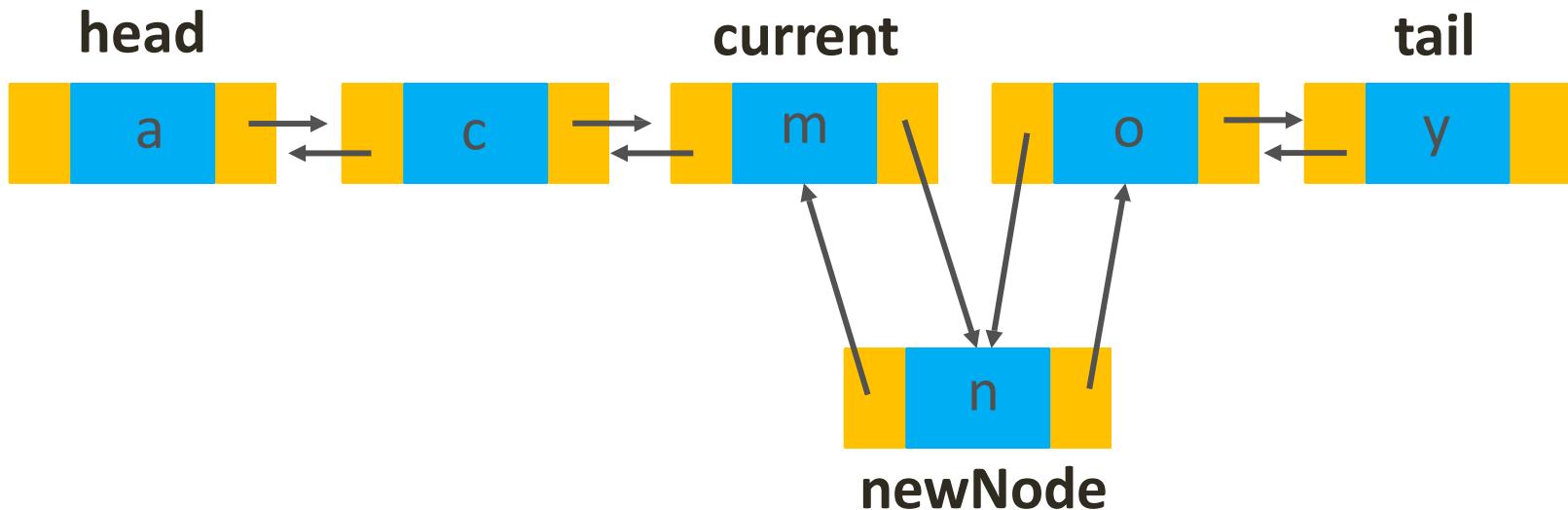
```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;  
newNode.next.prev = newNode;
```



# Doubly-linked list insertion

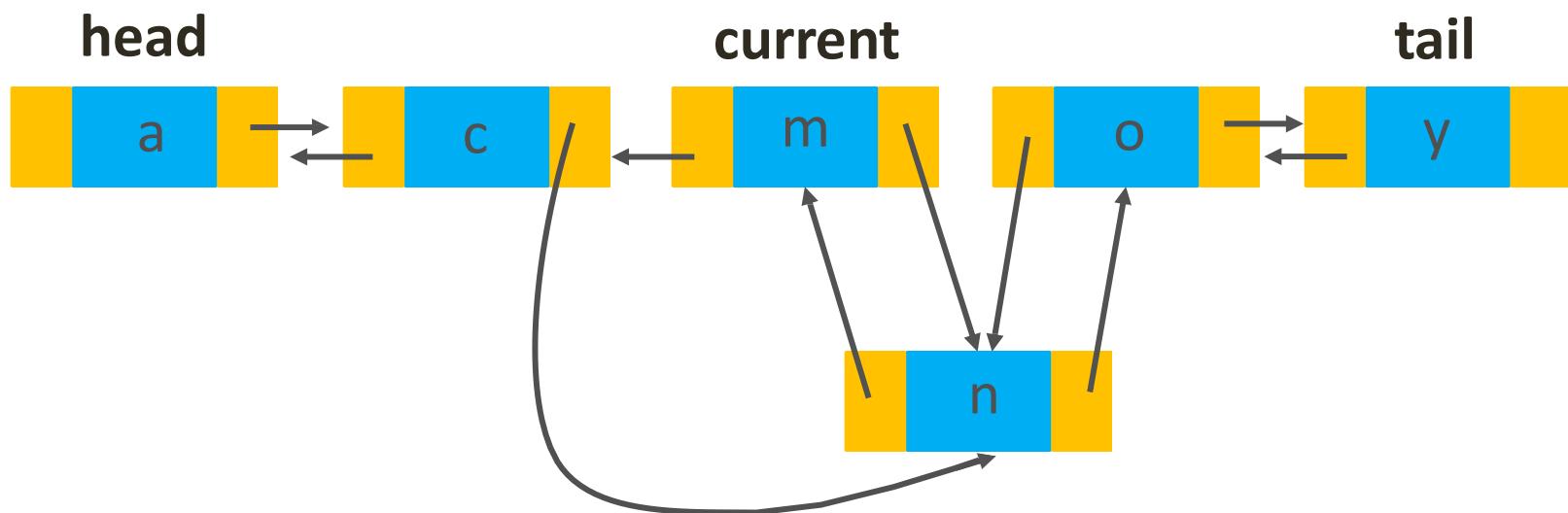
```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;  
newNode.next.prev = newNode;
```

1.  $c$   
2.  $\log N$   
3.  $N$   
4.  $N \log N$   
5.  $N^2$   
6.  $N^3$



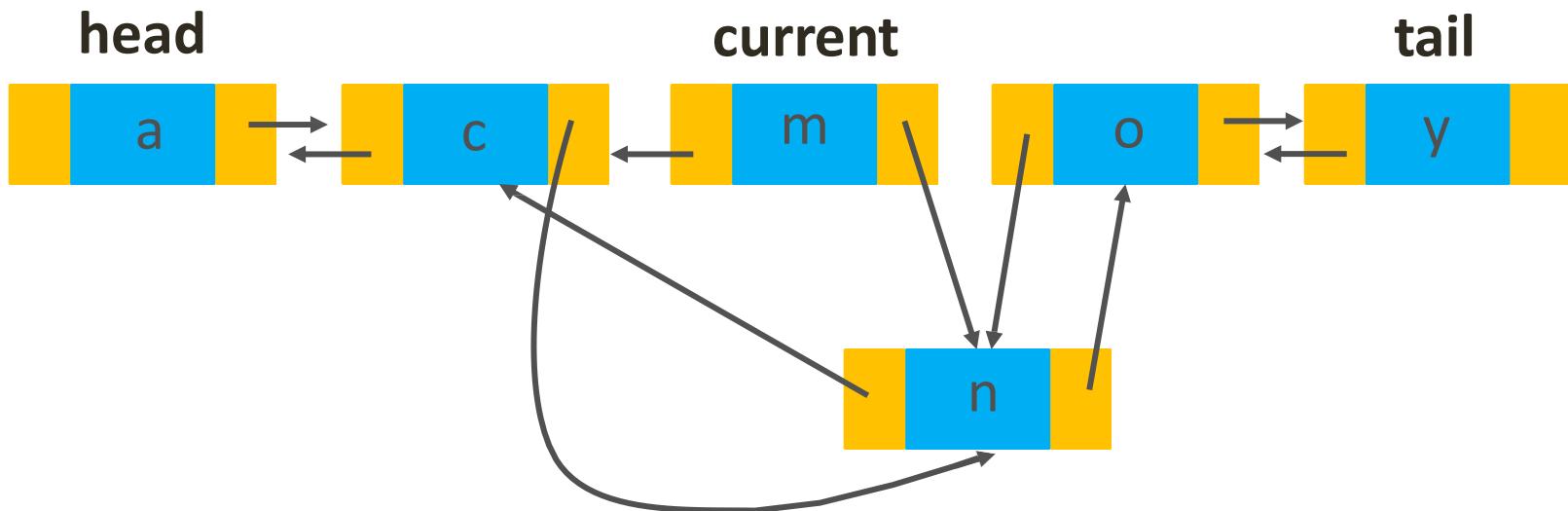
# Doubly-linked list deletion

```
current.prev.next = current.next;
```



# Doubly-linked list deletion

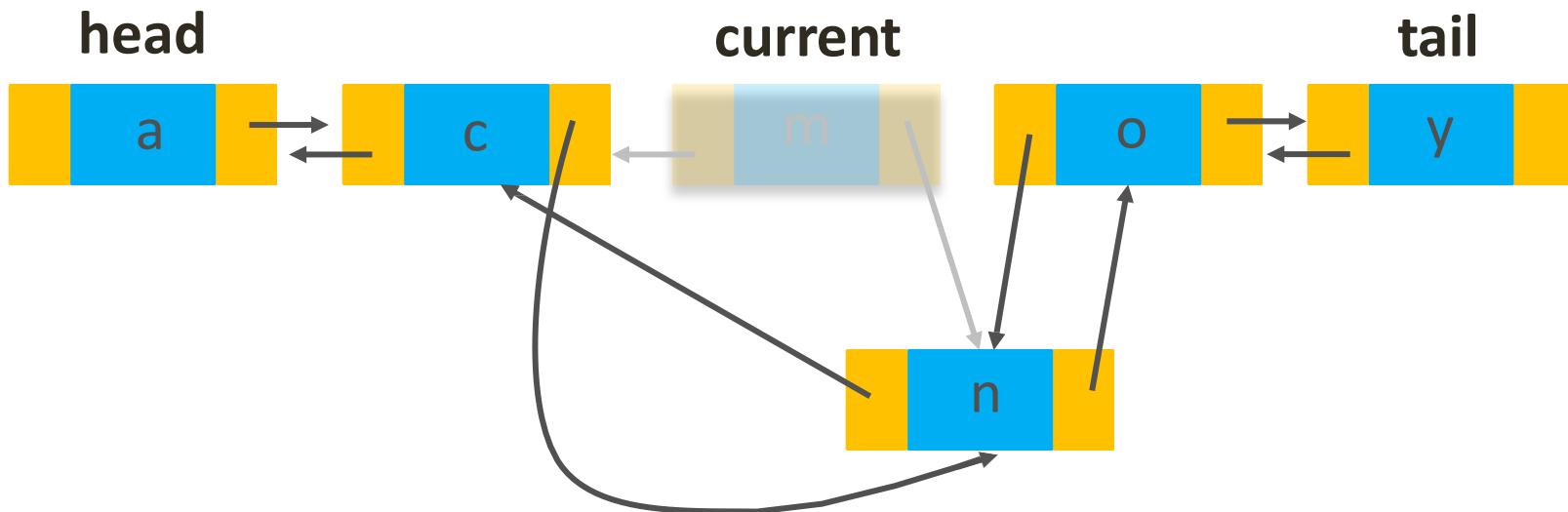
```
current.prev.next = current.next;  
current.next.prev = current.prev;
```



# Doubly-linked list deletion

```
current.prev.next = current.next;  
current.next.prev = current.prev;  
current.next = null;  
current.pre = null;
```

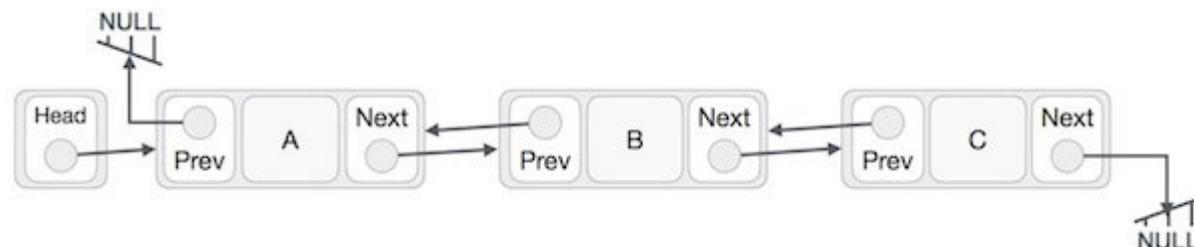
1. c
2. log N
3. N
4. N log N
5. N<sup>2</sup>
6. N<sup>3</sup>



```

1 package com.java2novice.ds.linkedlist;
2
3 import java.util.NoSuchElementException;
4
5 public class DoublyLinkedListImpl<E> {
6
7     private Node head;
8     private Node tail;
9     private int size;
10
11    public DoublyLinkedListImpl() {
12        size = 0;
13    }
14    /**
15     * this class keeps track of each element information
16     * @author java2novice
17     *
18     */
19    private class Node {
20        E element;
21        Node next;
22        Node prev;
23
24        public Node(E element, Node next, Node prev) {
25            this.element = element;
26            this.next = next;
27            this.prev = prev;
28        }
29    }
30    /**
31     * returns the size of the linked list
32     * @return
33     */
34    public int size() { return size; }
35
36    /**
37     * return whether the list is empty or not
38     * @return
39     */

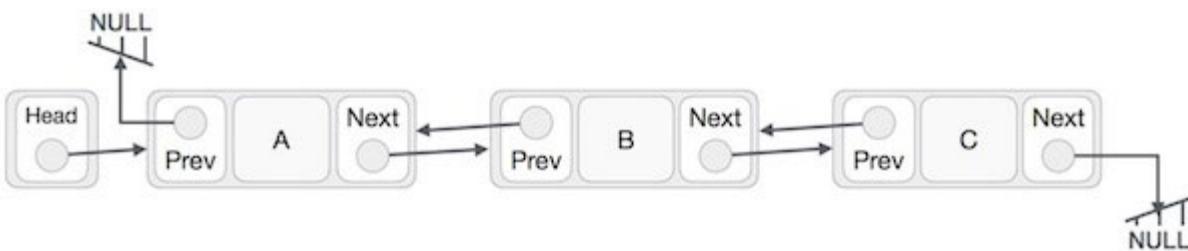
```



```

40 public boolean isEmpty() { return size == 0; }
41
42 /**
43 * adds element at the starting of the linked list
44 * @param element
45 */
46 public void addFirst(E element) {
47     Node tmp = new Node(element, head, null);
48     if(head != null) {head.prev = tmp;}
49     head = tmp;
50     if(tail == null) { tail = tmp;}
51     size++;
52     System.out.println("adding: "+element);
53 }
54
55 /**
56 * adds element at the end of the linked list
57 * @param element
58 */
59 public void addLast(E element) {
60
61     Node tmp = new Node(element, null, tail);
62     if(tail != null) {tail.next = tmp;}
63     tail = tmp;
64     if(head == null) { head = tmp;}
65     size++;
66     System.out.println("adding: "+element);
67 }
68
69 /**
70 * this method walks forward through the linked list
71 */
72 public void iterateForward(){
73
74     System.out.println("iterating forward..");
75     Node tmp = head;
76     while(tmp != null){
77         System.out.println(tmp.element);
78         tmp = tmp.next;
79     }
80 }

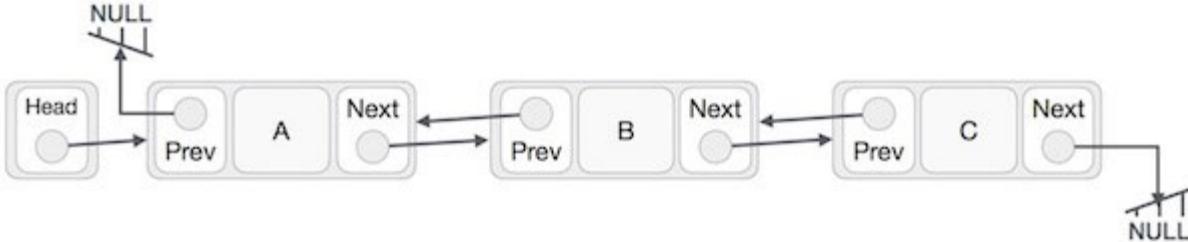
```



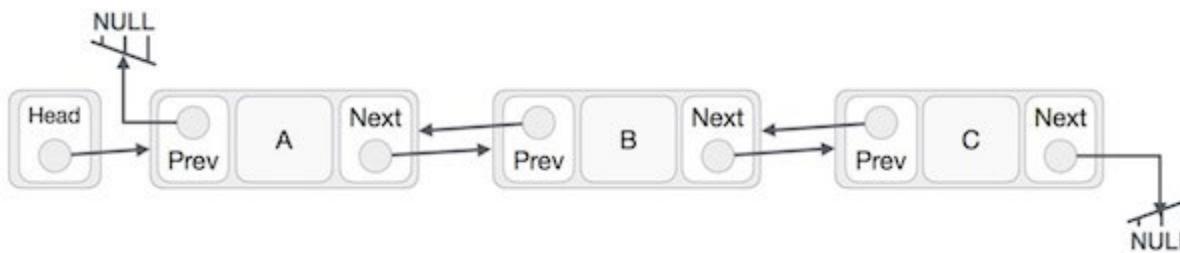
```

1 /**
2  * this method walks backward through the linked list
3 */
4 public void iterateBackward(){
5
6     System.out.println("iterating backword..");
7     Node tmp = tail;
8     while(tmp != null){
9         System.out.println(tmp.element);
10        tmp = tmp.prev;
11    }
12 }
13 /**
14  * this method removes element from the start of the linked list
15  * @return
16 */
17 public E removeFirst() {
18     if (size == 0) throw new NoSuchElementException();
19     Node tmp = head;
20     head = head.next;
21     head.prev = null;
22     size--;
23     System.out.println("deleted: "+tmp.element);
24     return tmp.element;
25 }
26 /**
27  * this method removes element from the end of the linked list
28  * @return
29 */
30 public E removeLast() {
31     if (size == 0) throw new NoSuchElementException();
32     Node tmp = tail;
33     tail = tail.prev;
34     tail.next = null;
35     size--;
36     System.out.println("deleted: "+tmp.element);
37     return tmp.element;
38 }

```



```
122  
123 public static void main(String a[]){  
124  
125     DoublyLinkedListImpl<Integer> dll = new DoublyLinkedListImpl<Integer>();  
126     dll.addFirst(10);  
127     dll.addFirst(34);  
128     dll.addLast(56);  
129     dll.addLast(364);  
130     dll.iterateForward();  
131     dll.removeFirst();  
132     dll.removeLast();  
133     dll.iterateBackward();  
134 }  
135 }
```



# Things to consider...

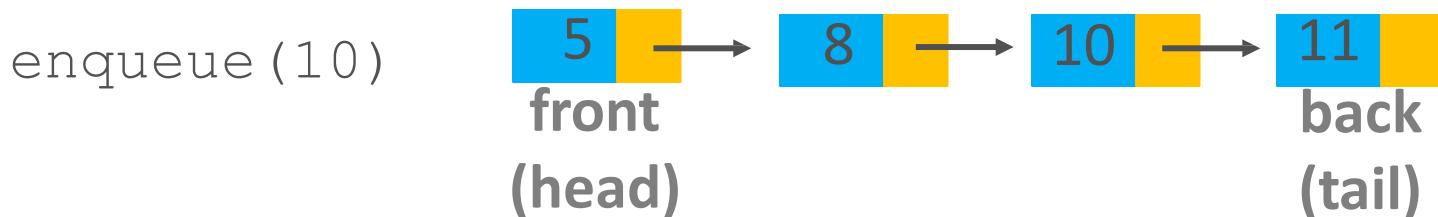
- Adding to the front or end of a linked list is a little different than adding somewhere in the middle
  - why?
- Removing from a list with 1 node
  - what happens to head/tail?
- Adding to an empty list
  - what is the current value of head/tail?

# Priority queue

- Like a queue, but items returned in order of ***priority***
  - *Dequeue* operation always returns the item with the highest priority
  - If two items have the same priority, the first one in the queue is returned
- Example?!

# Priority queue as a linked list...

- Always add items in correct, sorted spot



- dequeue will return smallest item  $O(1)$
- What is the cost of enqueue?
- We will study a more advanced priority queue later...

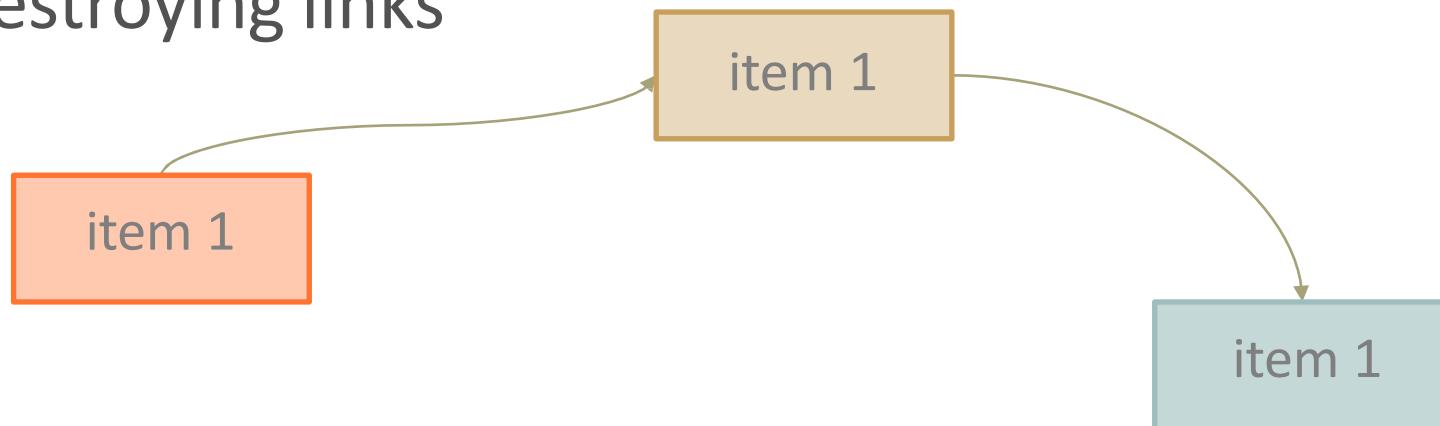


# Trees

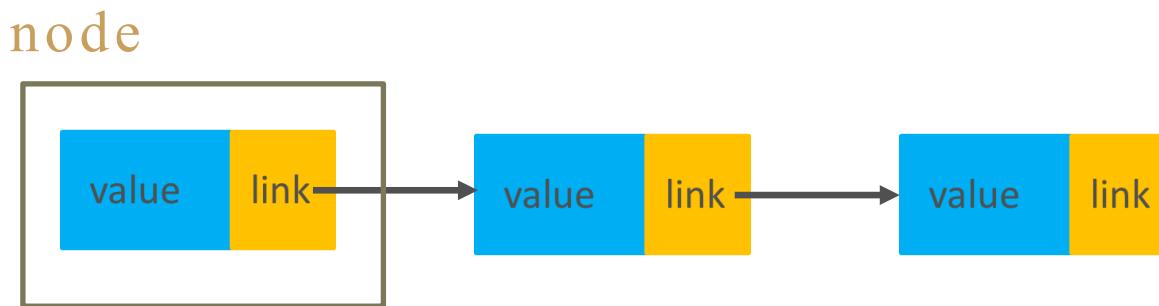
## CSC220|Computer Programming 2

Last Time...

- **Linked structures**
  - Data storage structure
  - Individual items have *links* (references) to other items
- Items don't reside in a single contiguous block of memory
- Items can be *dynamically* added or removed from the structure, simply by creating or destroying links



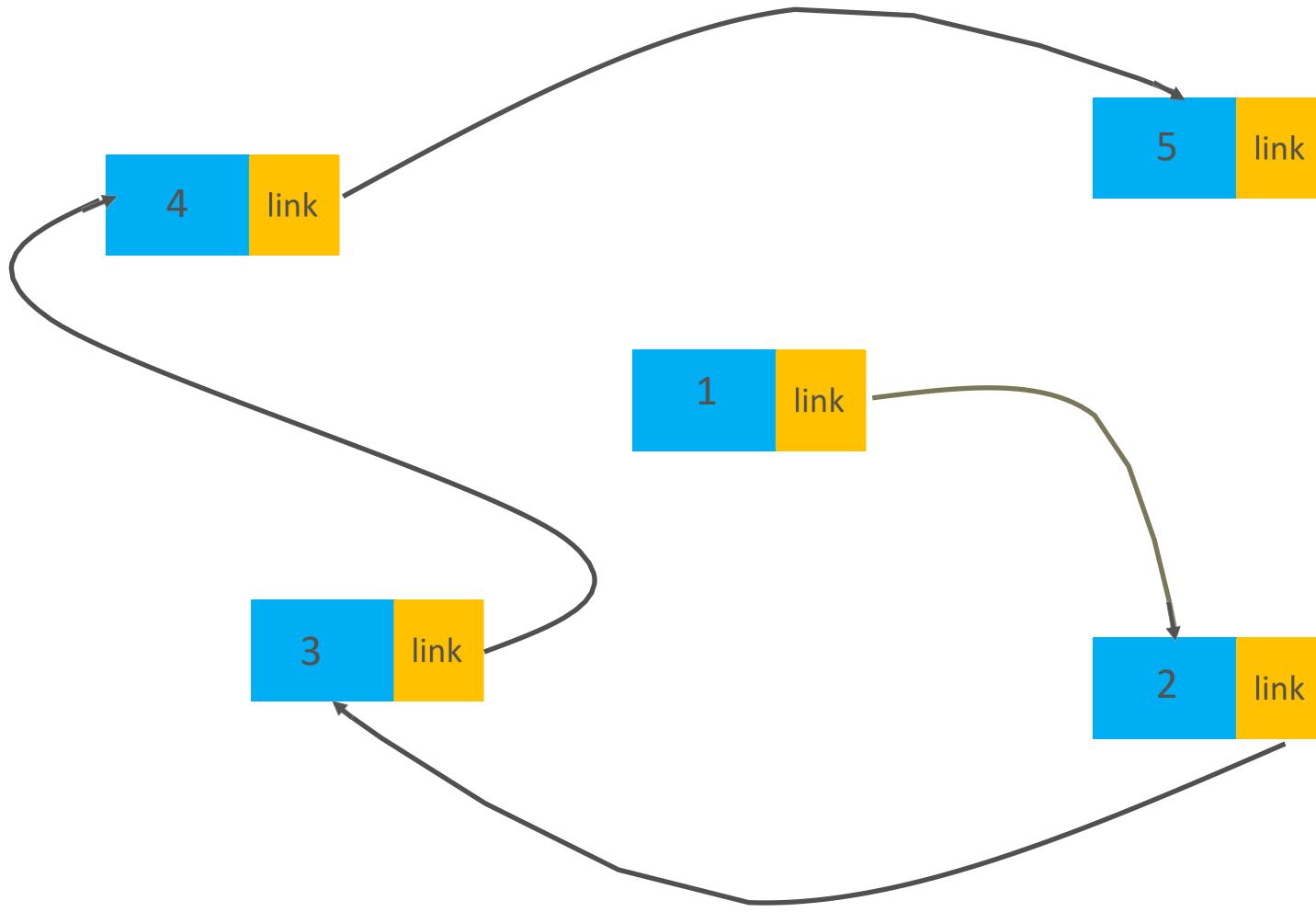
- A **linked list** is another way to implement a list
- Each **node**, or item in the list, has a link to the next item in the list



- A single node consists of some **data** and a **reference** to another node

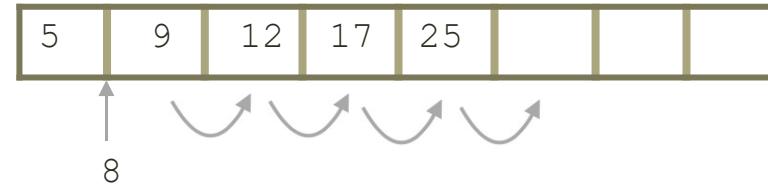
LinkedList<E>

- Nodes may not be contiguous in memory!



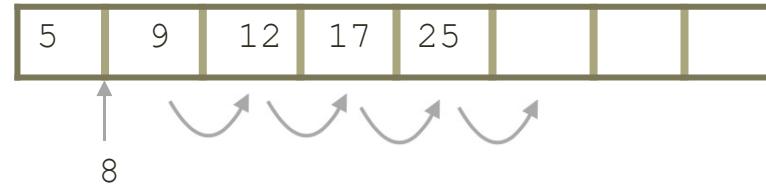
# Insertion and Deletion

- Insertion into an array:



- Insert into an array list

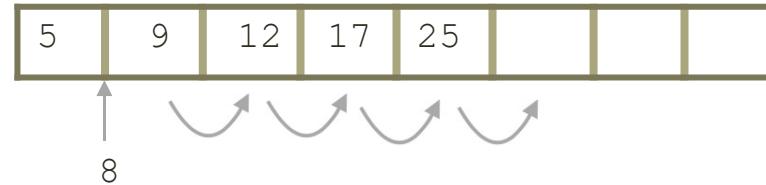
- Insertion into an array:



- Insert into an array list



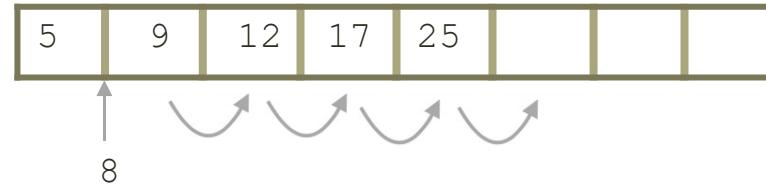
- Insertion into an array:



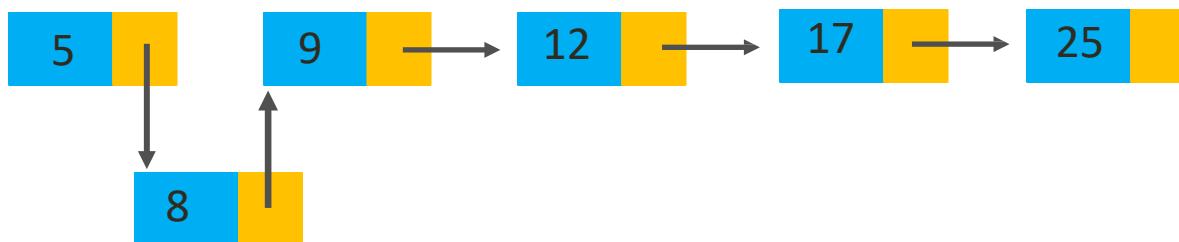
- Insert into an array list



- Insertion into an array:



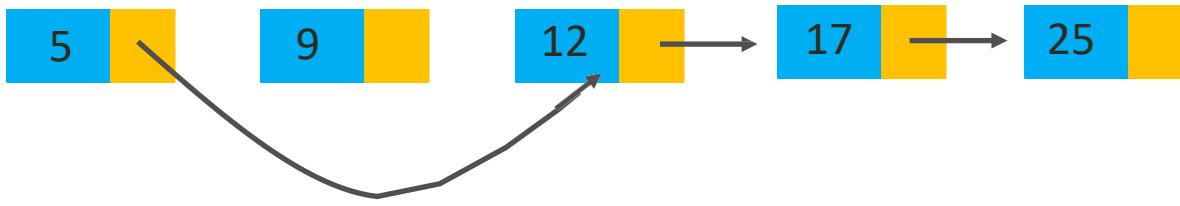
- Insert into an array list



# Deletion from a linkedlist



# Deletion from a linkedlist



# Deletion from a linkedlist

- Deletion from a linked list



9 is now stranded  
garbage collector will clean it up

# Traverse a linked list

|                         | Array                    | Linked list                              |
|-------------------------|--------------------------|------------------------------------------|
| Go to the front         | <code>int i = 0;</code>  | <code>LinkedListNode temp = head;</code> |
| Test for more elements  | <code>i &lt; size</code> | <code>temp != null</code>                |
| Get the current value   | <code>array[i]</code>    | <code>temp.data</code>                   |
| Got to the next element | <code>i++;</code>        | <code>temp = temp.next;</code>           |

# Appending add



```
LinkedNode current = head;
```

```
While (current.next != null)
```

```
    current = current.next;
```

```
current.next = new LinkedNode(30);
```

# Doubly-linked list

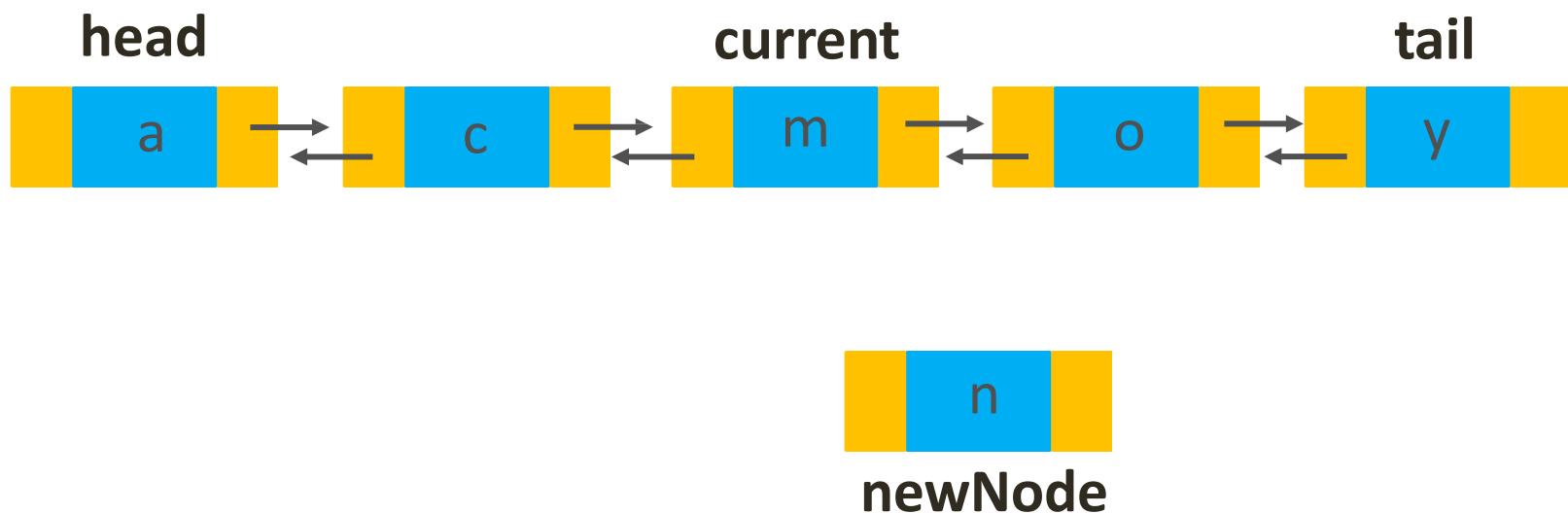
- Nodes have a link to `next` *and* `previous` node
- Allows for traversal in either forward or reverse order
- Maintains a `tail` node as well as a `head` node

# Doubly-linked list insertion



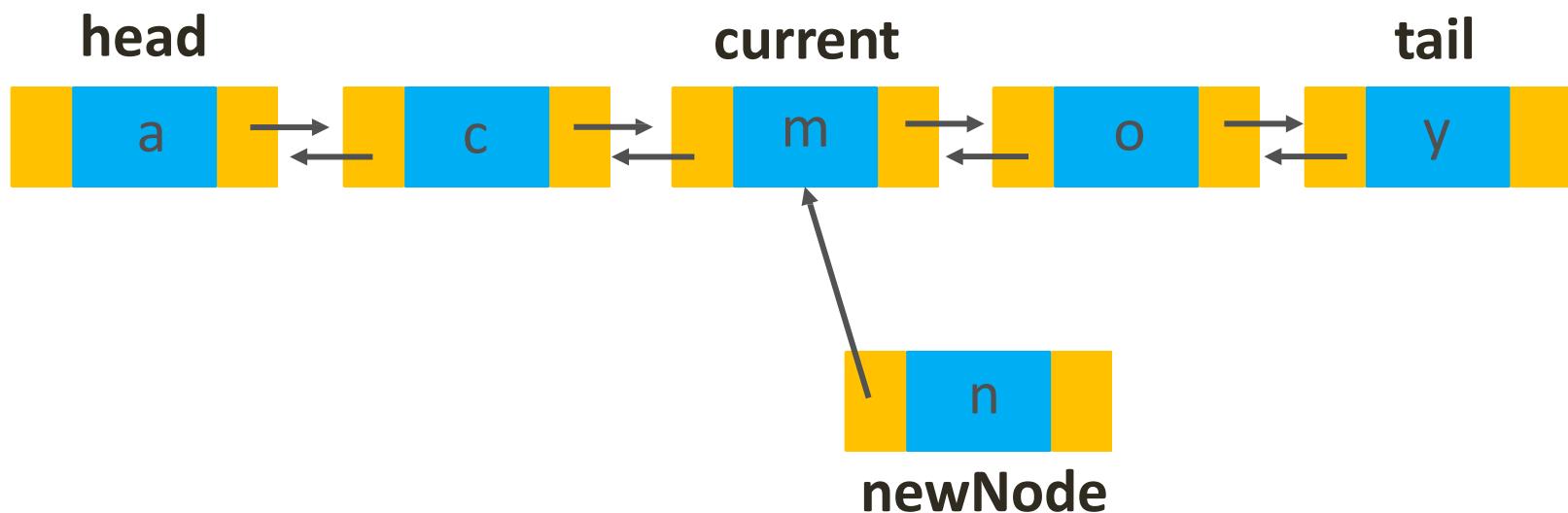
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';
```



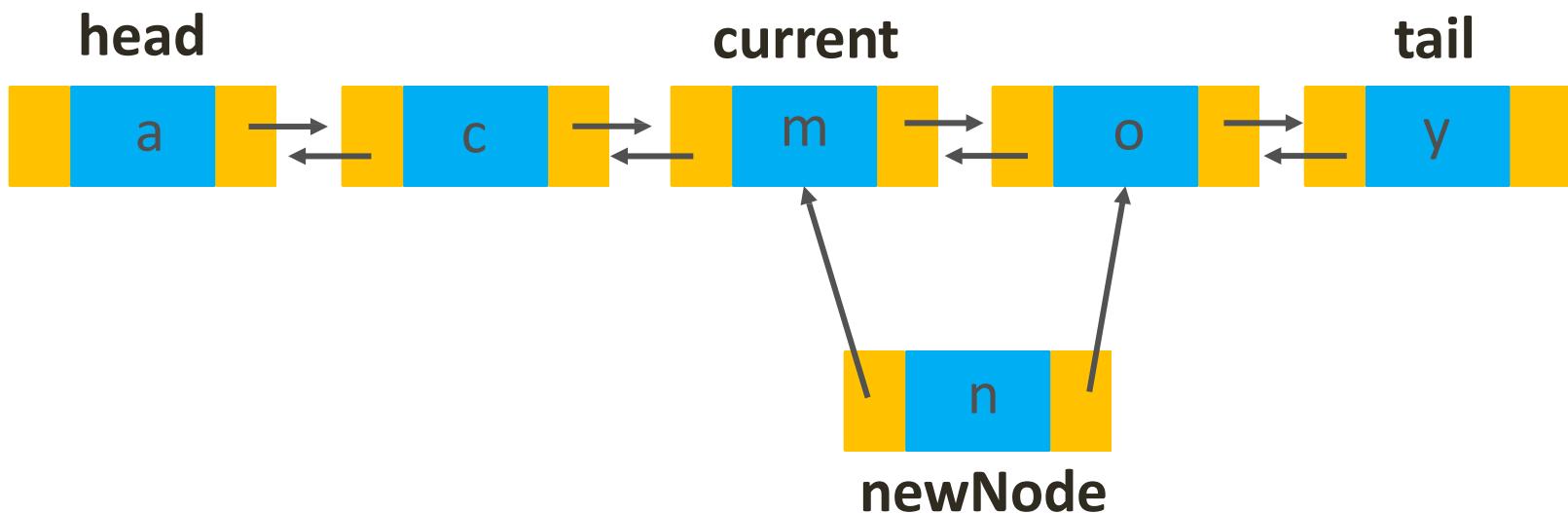
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;
```



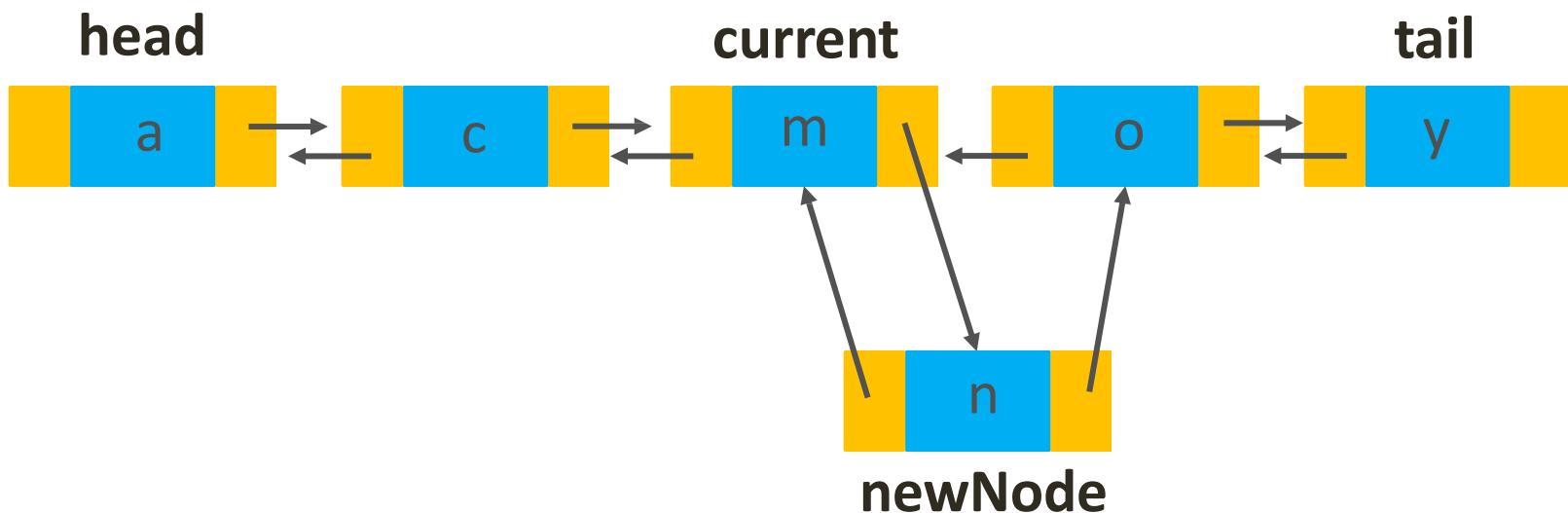
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;
```



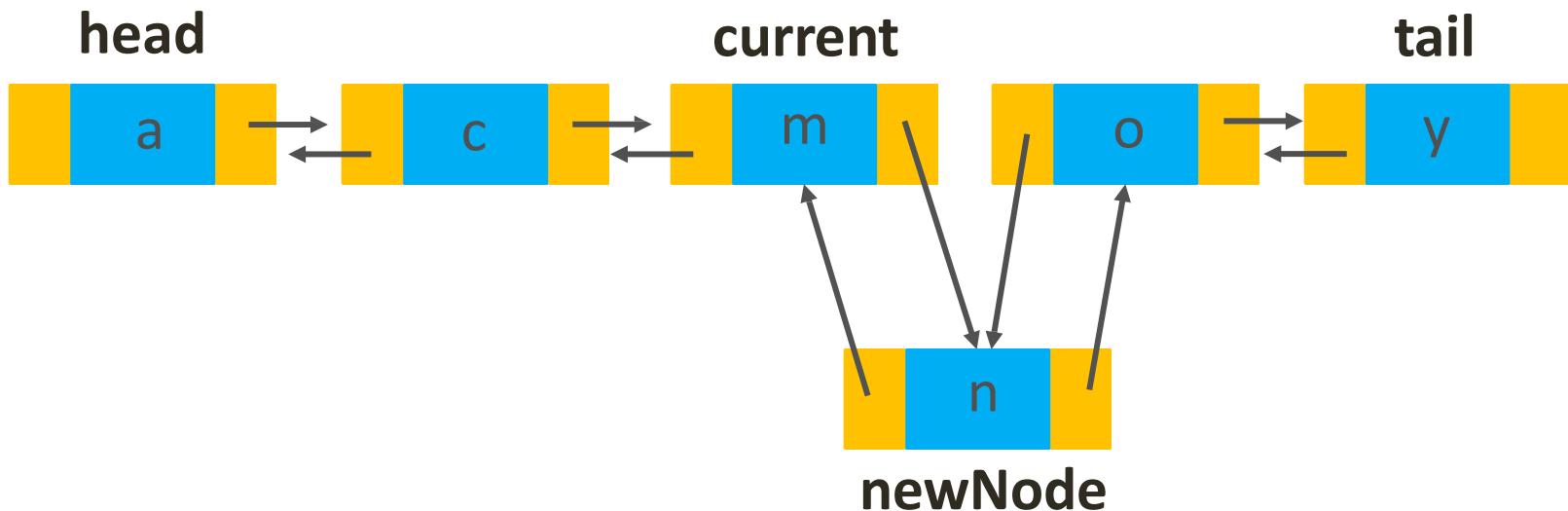
# Doubly-linked list insertion

```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;
```



# Doubly-linked list insertion

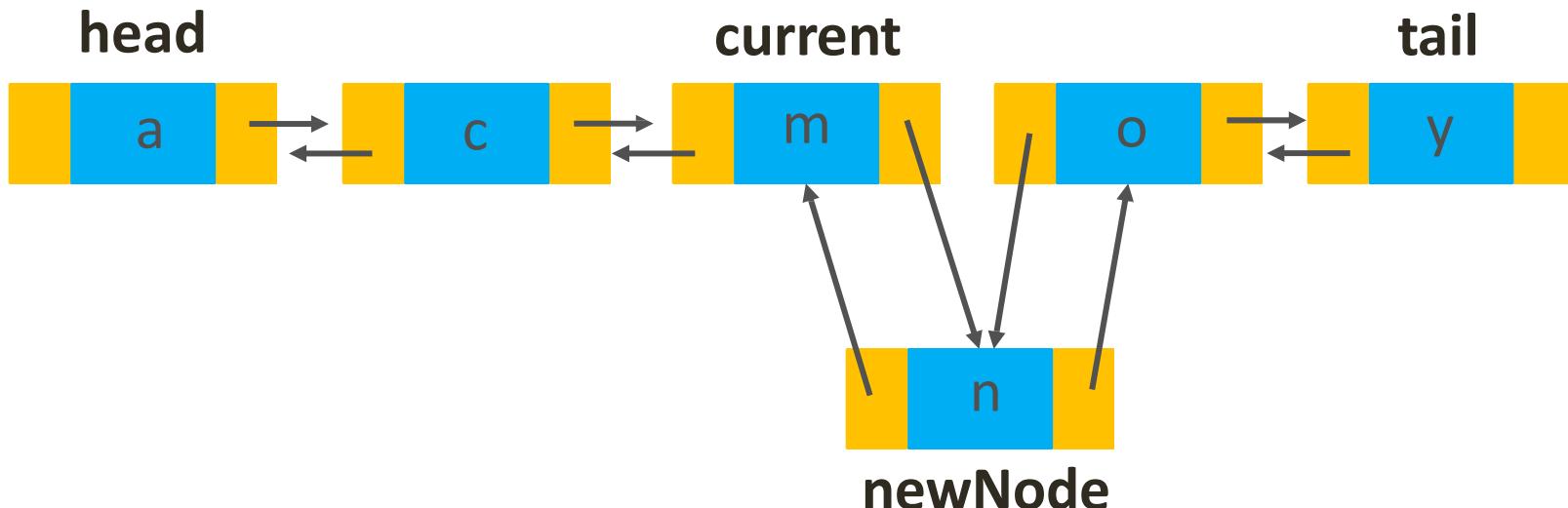
```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;  
newNode.next.prev = newNode;
```



# Doubly-linked list insertion

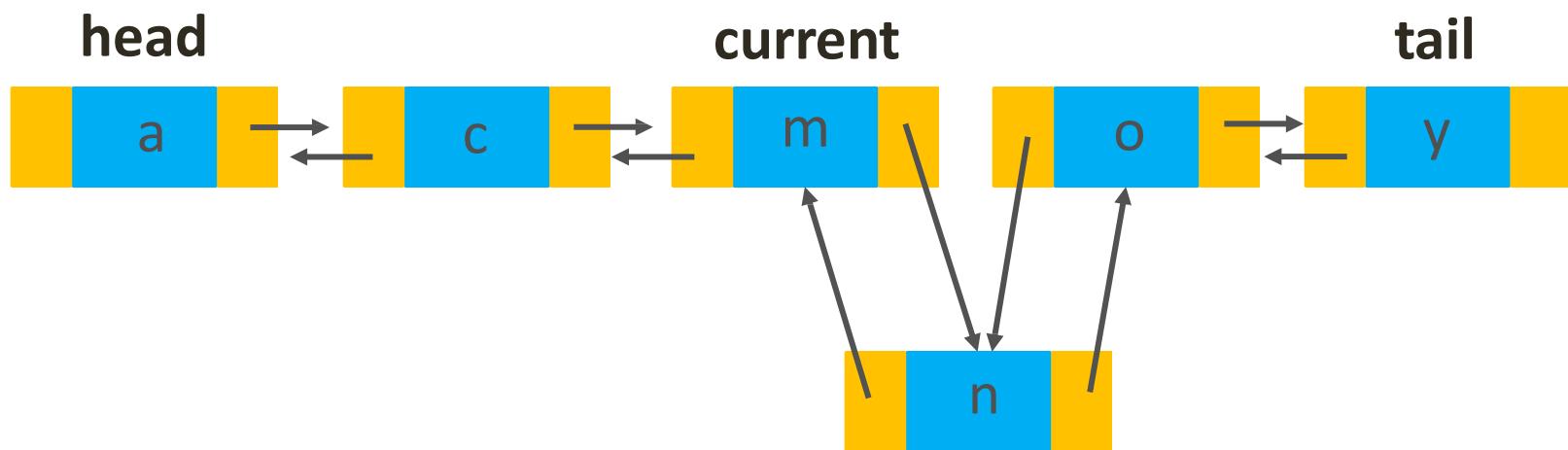
```
newNode = new LinkedNode<Character>();  
newNode.data = 'n';  
  
newNode.prev = current;  
newNode.next = current.next;  
newNode.prev.next = newNode;  
newNode.next.prev = newNode;
```

1.  $c$   
2.  $\log N$   
3.  $N$   
4.  $N \log N$   
5.  $N^2$   
6.  $N^3$



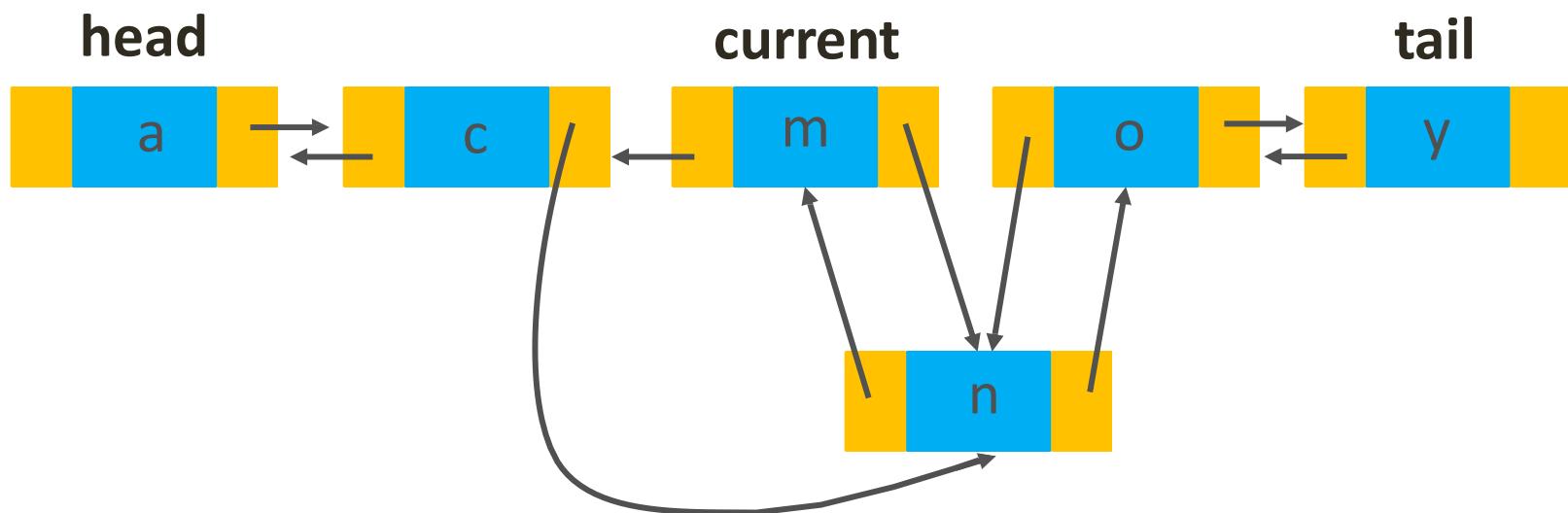
# Doubly-linked list deletion

```
current.prev.next = current.next;  
current.next.prev = current.prev;
```



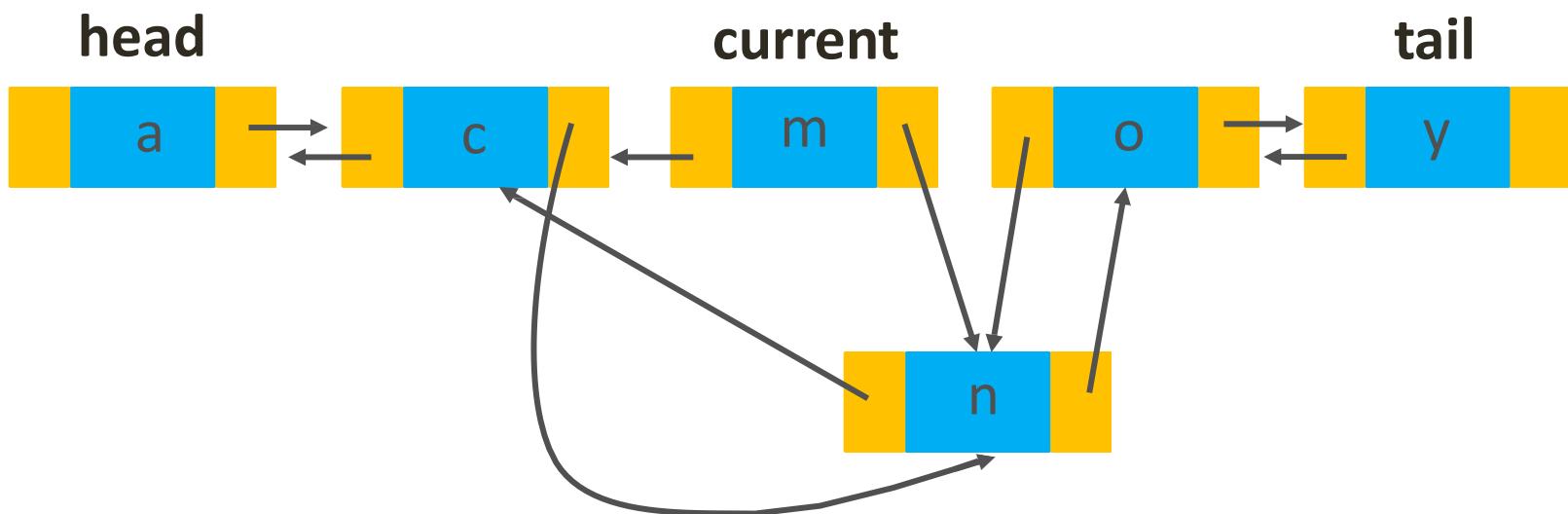
# Doubly-linked list deletion

```
current.prev.next = current.next;
```



# Doubly-linked list deletion

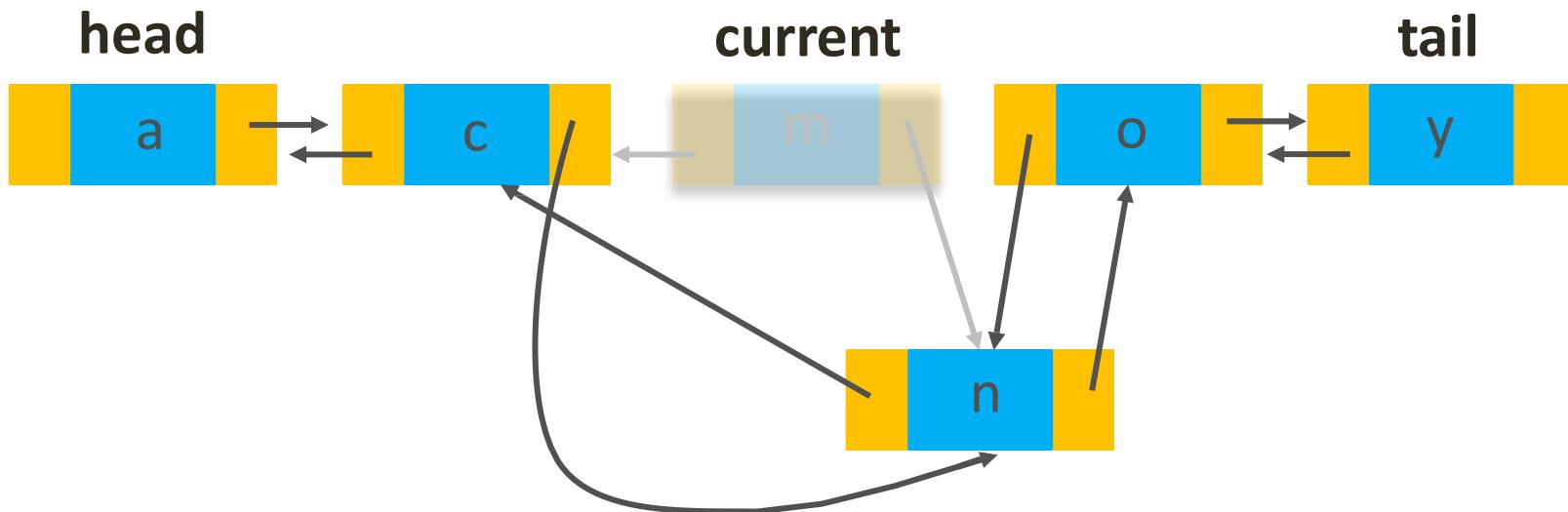
```
current.prev.next = current.next;  
current.next.prev = current.prev;
```



# Doubly-linked list deletion

```
current.prev.next = current.next;  
current.next.prev = current.prev;  
current.next = null;  
current.pre = null;
```

1. c
2. log N
3. N
4. N log N
5. N<sup>2</sup>
6. N<sup>3</sup>

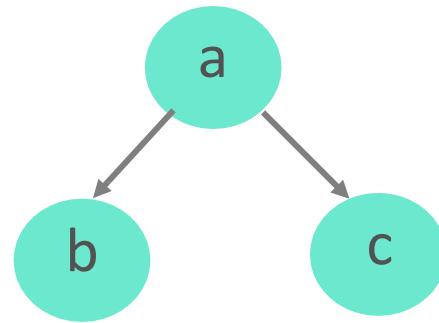


Today...

# Trees

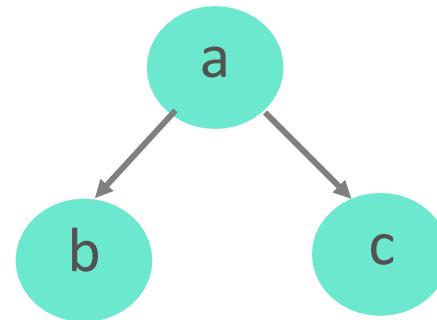
# Tree

- A linked data structure with a hierarchical formation



# Tree

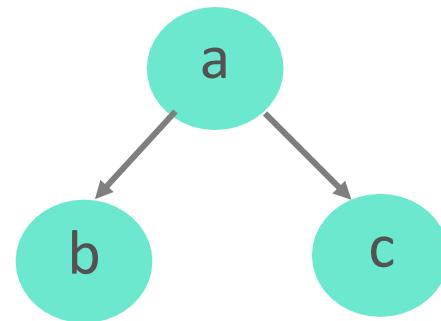
- A linked data structure with a hierarchical formation
- Trees can have multiple links, called branches



# Tree

- A linked data structure with a hierarchical formation
- Trees can have multiple links, called branches

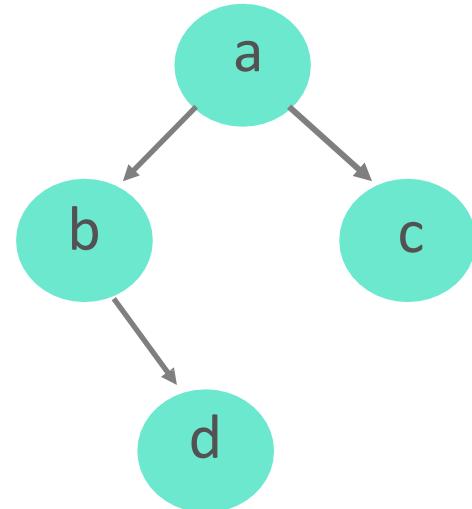
There are multiple directions  
you can take at any  
given node



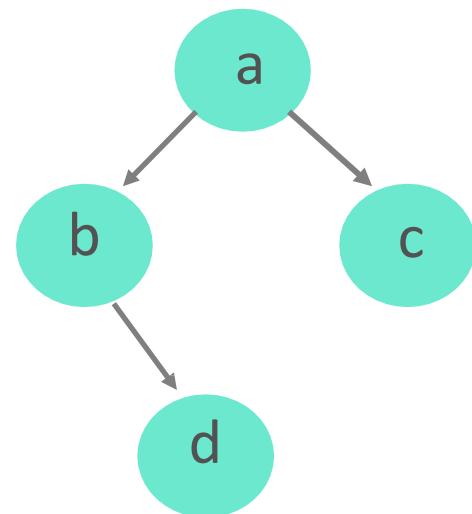
- Trees have a **hierarchical structure**
  - Meaning, any node is a subtree of some larger tree
    - *Except the very top node!*
  - In CS, trees are usually represented with the root at the top

- Trees have a **hierarchical structure**
  - Meaning, any node is a subtree of some larger tree
    - *Except the very top node!*
  - In CS, trees are usually represented with the root at the top
- Trees are **recursive** in nature
  - Any given node (containing its descendant nodes) is itself a tree
  - A tree consists of:
    - *A data element...*
    - *...and more subtrees*

- There is a strict parent-to-child relationship among nodes
  - Links *only* go from parent to child
    - *Not from child to parent*
    - *Not from sibling to sibling*

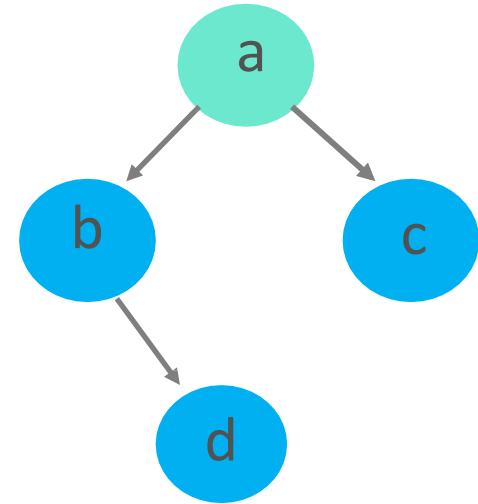


- There is a strict parent-to-child relationship among nodes
  - Links *only* go from parent to child
    - *Not from child to parent*
    - *Not from sibling to sibling*
- Every node has exactly **one** parent, except for the **root**, which has none
- There is exactly one path from the root to any other node

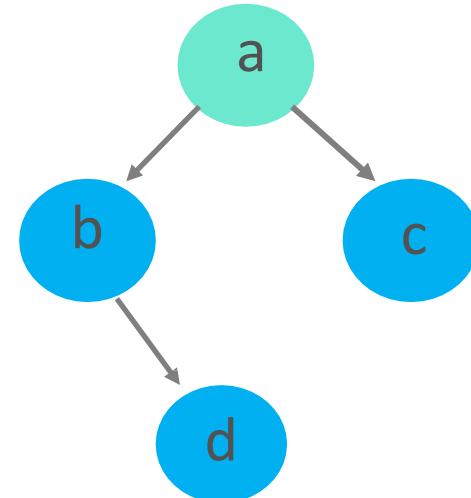


# Terminology

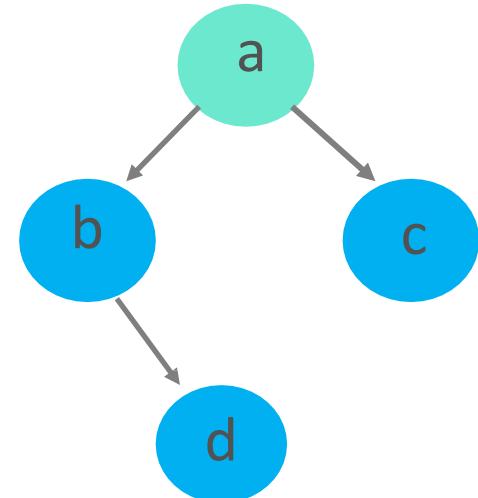
- **Root node:** the single node in a tree that has no parents



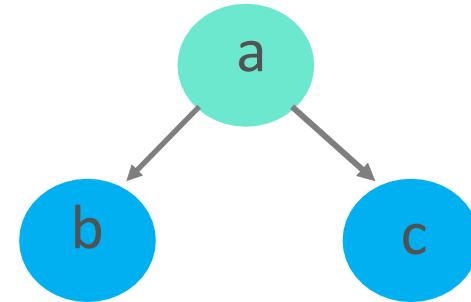
- **Root node:** the single node in a tree that has no parents
- **Parent:** a node's parent has a direct reference to it
  - nodes have AT MOST one parent



- **Root node:** the single node in a tree that has no parents
- **Parent:** a node's parent has a direct reference to it
  - nodes have AT MOST one parent
- **Child:** a node B is a child of node A if A has a direct reference to B



- **Root node:** the single node in a tree that has no parents
- **Parent:** a node's parent has a direct reference to it
  - nodes have AT MOST one parent
- **Child:** a node B is a child of node A if A has a direct reference to B
- **Sibling:** two nodes are siblings if they have the same parent

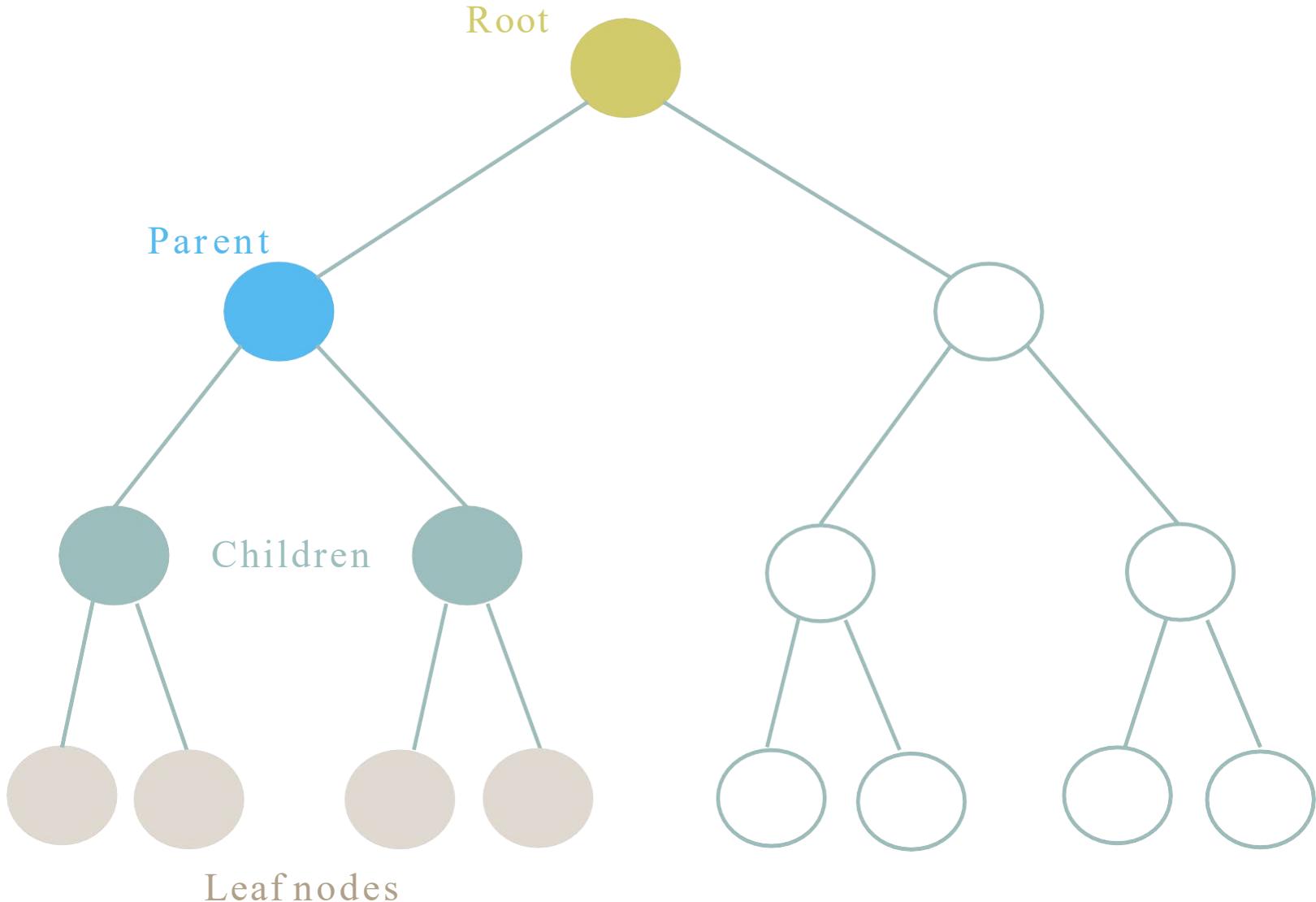


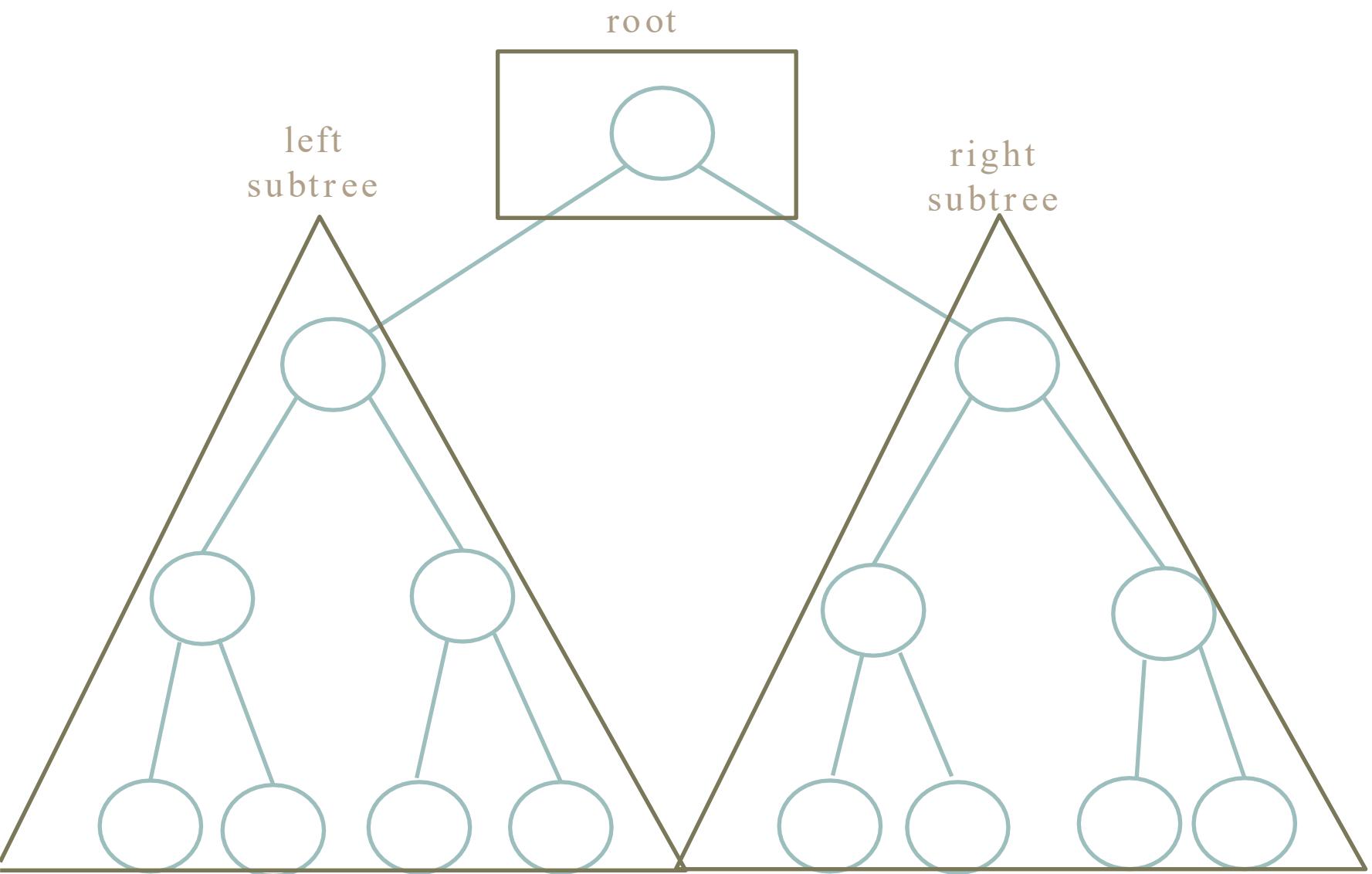
- **Leaf node:** a node with no children

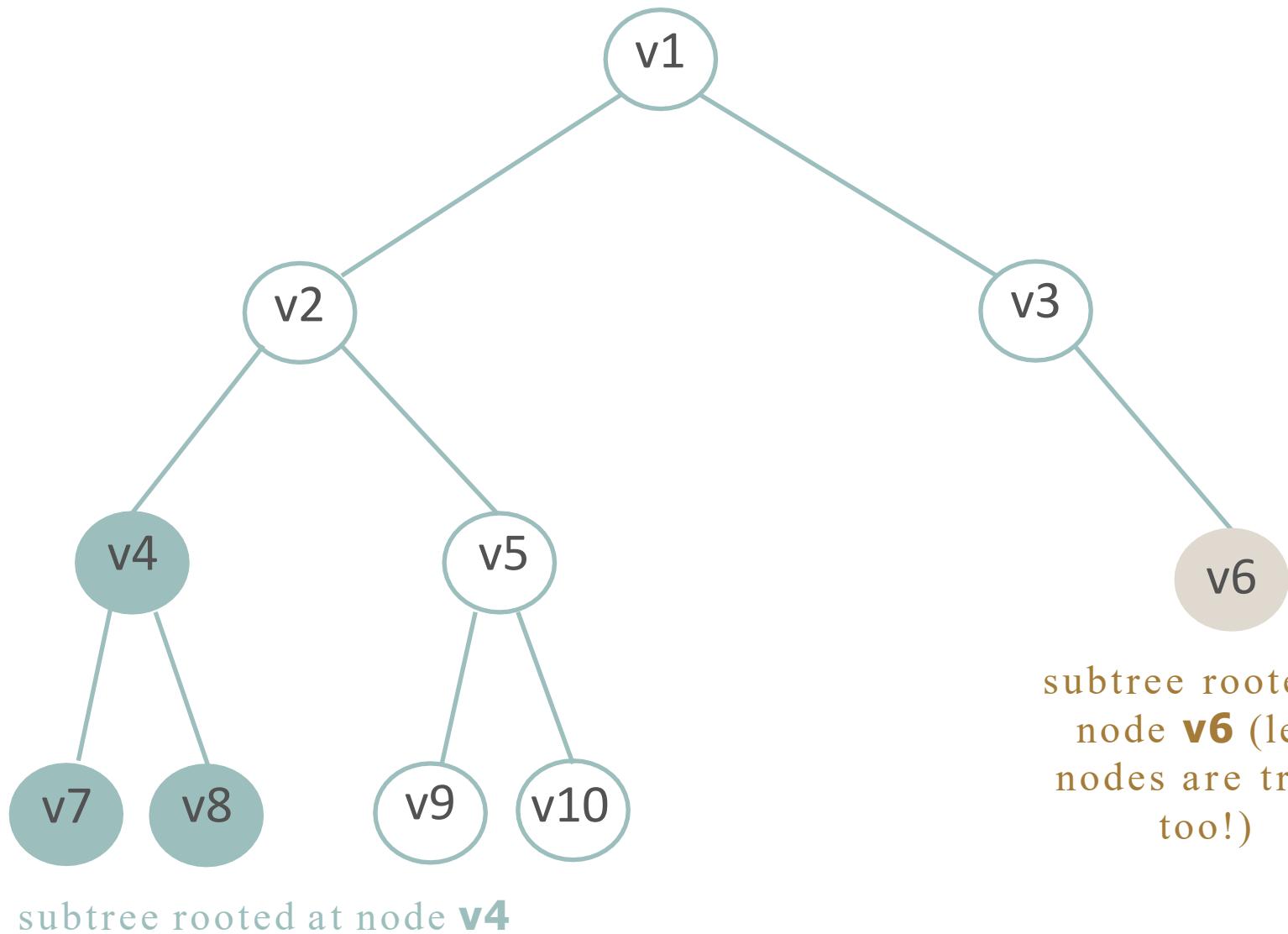
- **Leaf node:** a node with no children
- **Inner node:** a node with at least one child

- **Leaf node:** a node with no children
- **Inner node:** a node with at least one child
- **Depth:** the number of ancestors a node has
  - i.e., how many steps to the root
  - children are exactly one level deeper than their parents
  - a root node has depth 0

- **Leaf node:** a node with no children
- **Inner node:** a node with at least one child
- **Depth:** the number of ancestors a node has
  - i.e., how many steps to the root
  - children are exactly one level deeper than their parents
  - a root node has depth 0
- **Height:** the depth of a tree's deepest leaf node + 1



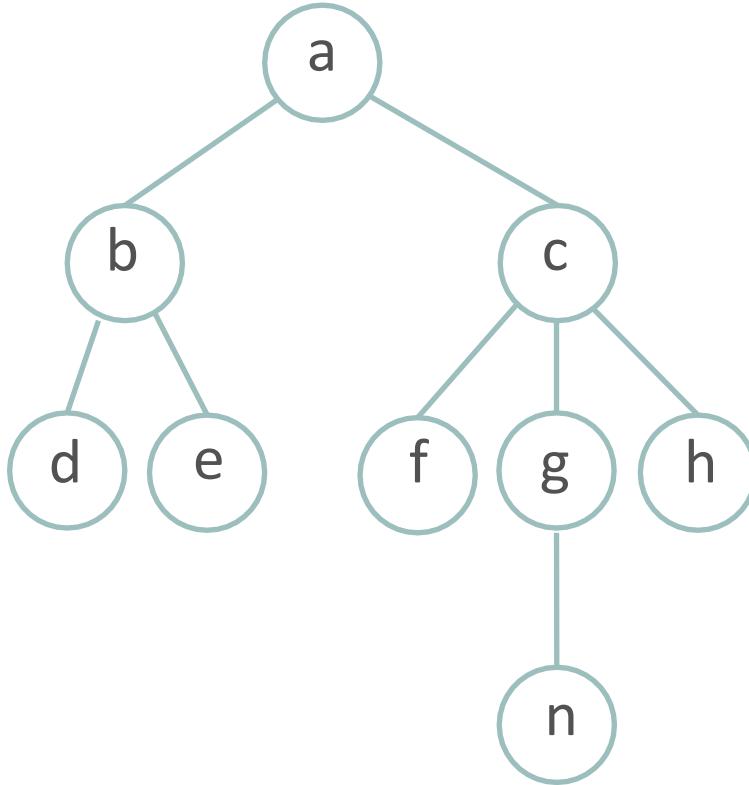




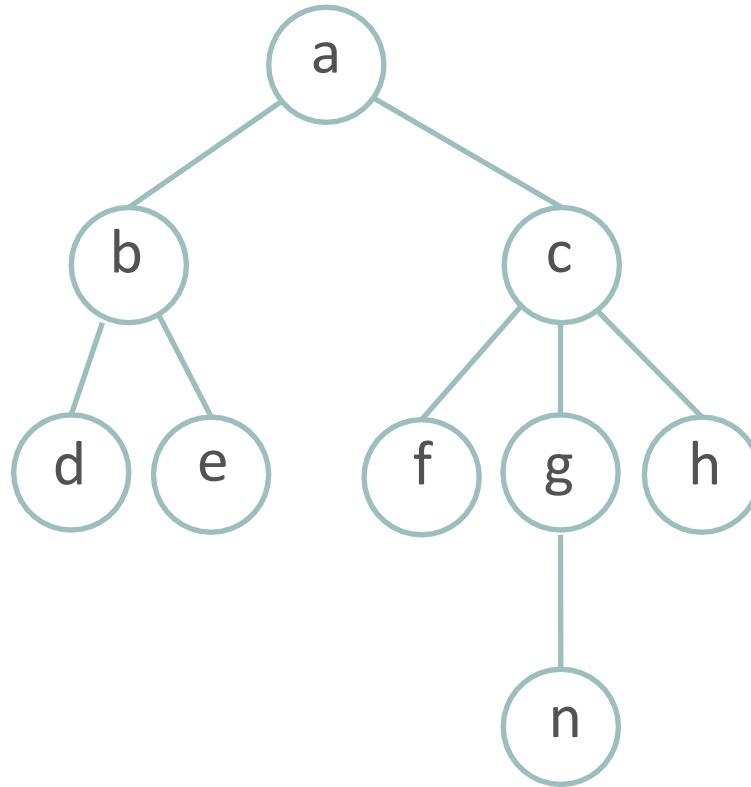
subtree rooted at  
node **v6** (leaf  
nodes are trees  
too!)

# Example

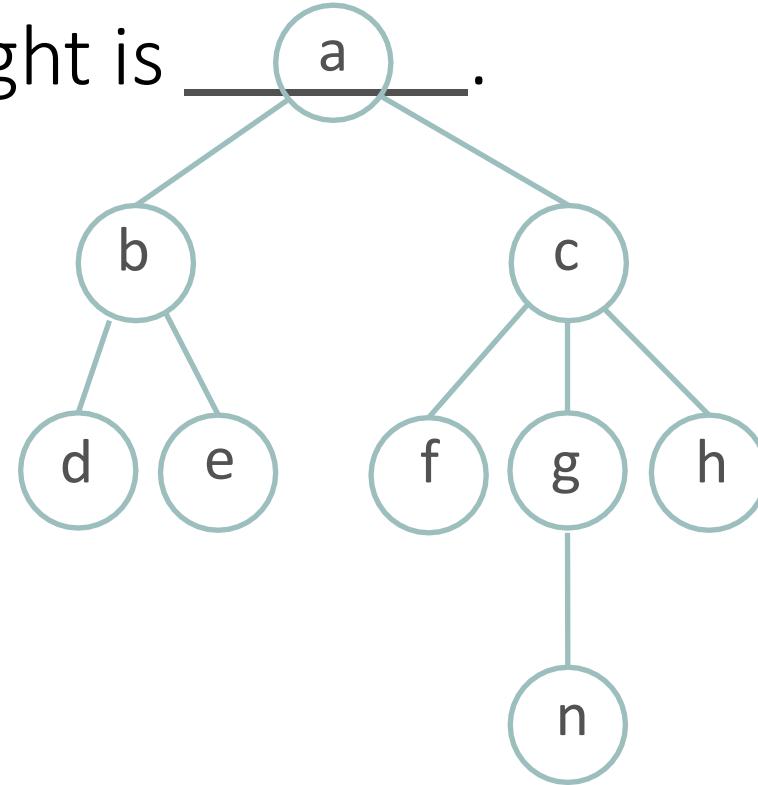
The root is \_\_\_\_.



The root is a.  
The height is 4  
\_\_\_\_\_.

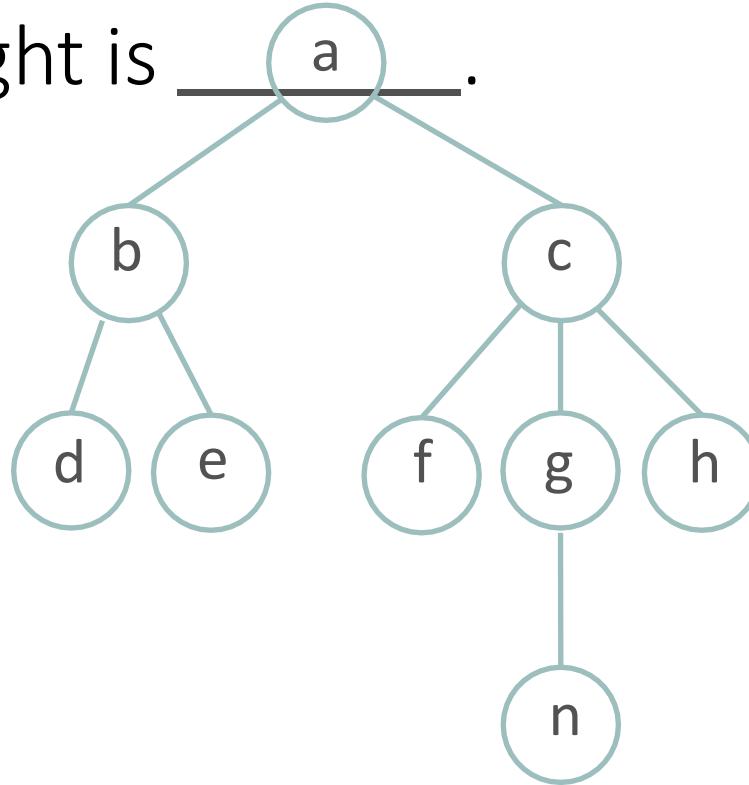


The root is \_\_\_. The height is \_\_\_.  
The parent of **e** is \_\_\_.



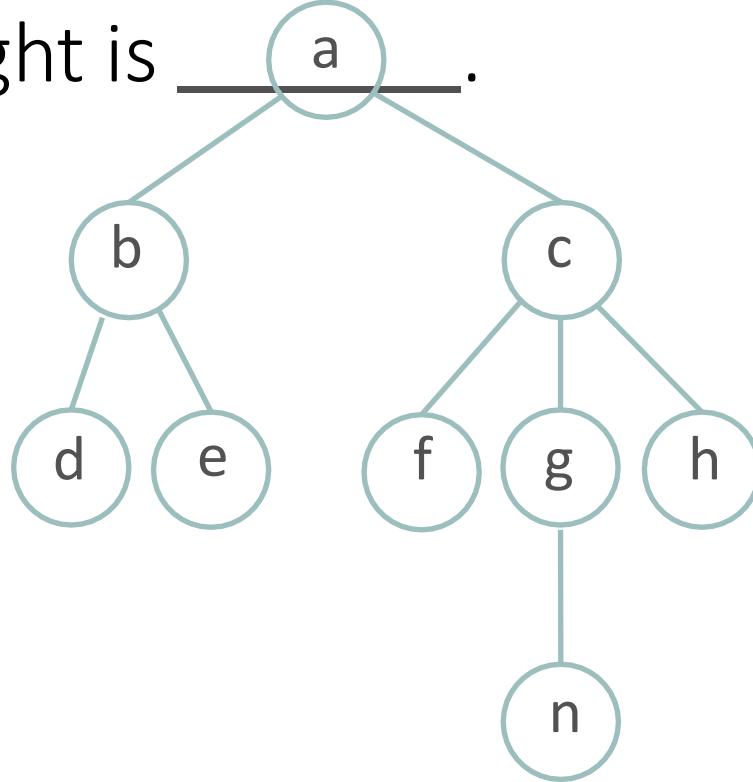
The root is \_\_\_. The height is \_\_\_.  
The parent of **e** is \_\_\_.

The depth of **e** is \_\_\_.



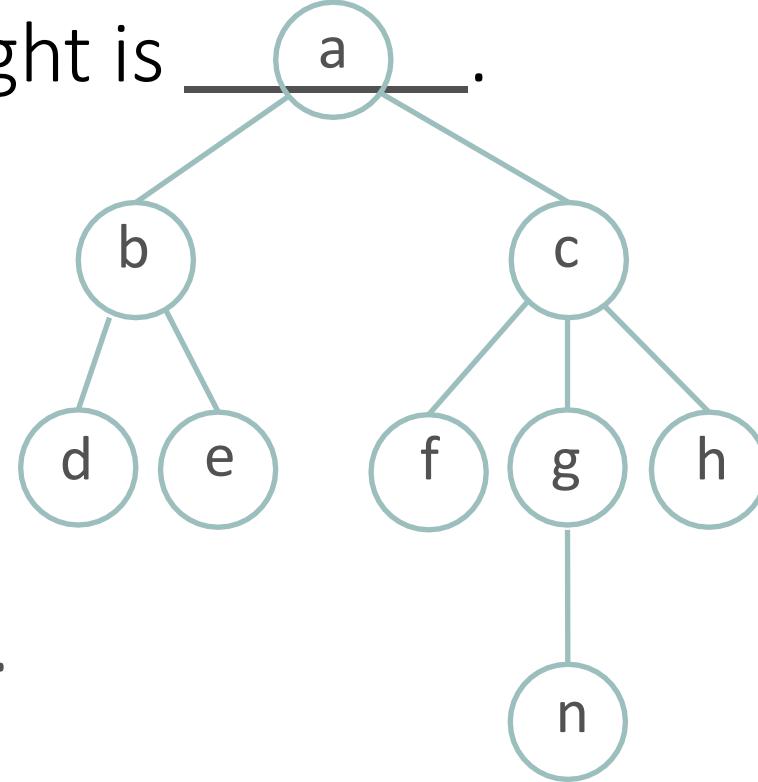
The root is \_\_\_. The height is \_\_\_\_.  
The parent of **e** is \_\_\_\_.

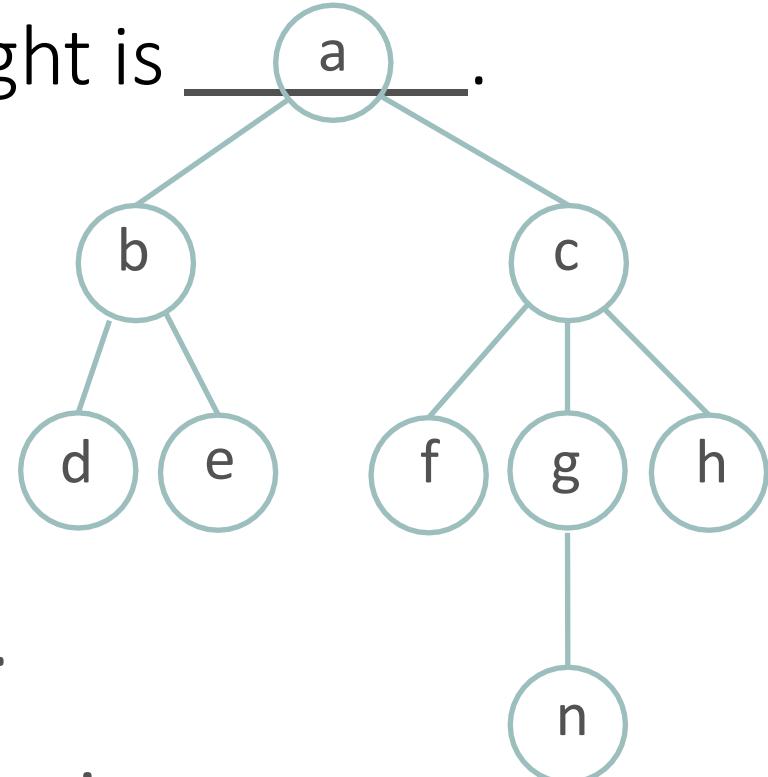
The depth of **e** is \_\_\_\_.  
The children of **c** are \_\_\_\_.



The root is \_\_\_. The height is \_\_\_.  
The parent of **e** is \_\_\_.

The depth of **e** is \_\_\_.  
The children of **c** are \_\_\_.  
The ancestors of **d** are \_\_\_.



The root is \_\_\_\_\_. The height is \_\_\_\_\_.  
The parent of **e** is \_\_\_\_\_.  
  


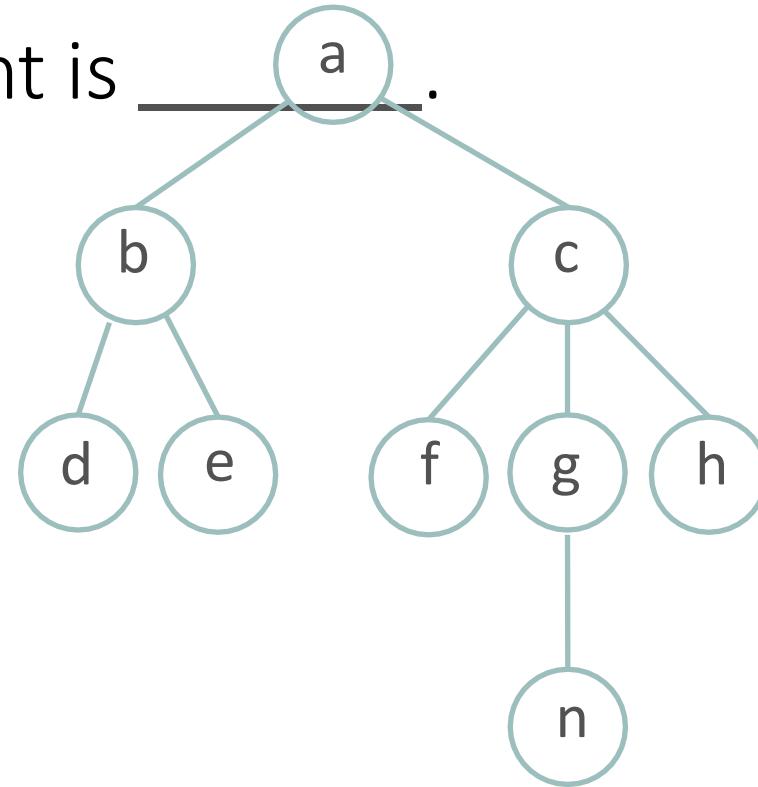
The depth of **e** is \_\_\_\_\_.

The children of **c** are \_\_\_\_\_.

The ancestors of **d** are \_\_\_\_\_.

The descendants of **c** are \_\_\_\_\_.

The root is \_\_\_. The height is \_\_\_.  
The parent of **e** is \_\_\_.



The leaves are \_\_\_

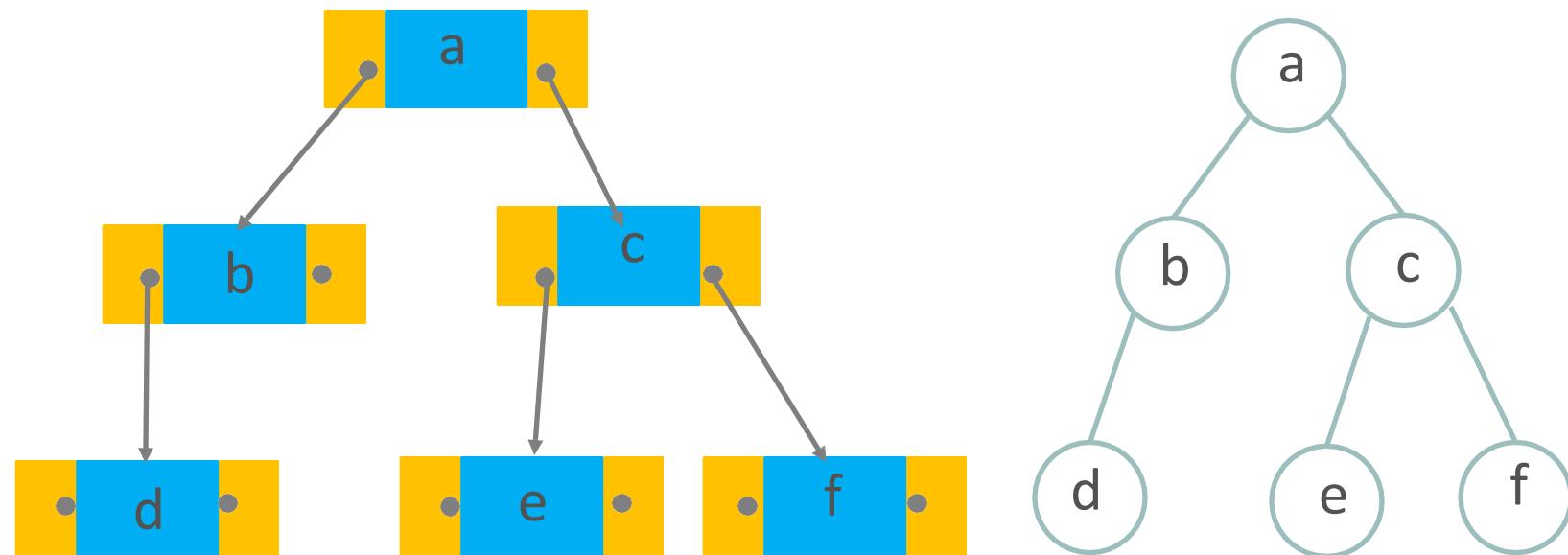
# Binary Trees

- **Binary trees** are a special case of a tree in which a node can have **AT MOST** two children
- These nodes are designated *left* and *right*
- In this class we will mostly concentrate on binary trees

- **Binary trees** are a special case of a tree in which a node can have **AT MOST** two children
- These nodes are designated *left* and *right*
- In this class we will mostly concentrate on binary trees

What should the implementation of a  
binary tree look like?  
What about a binary tree node?

- Each node has two reference variables
  - One for each of the two children
- If there is no child, the reference is set to null



```
class BinaryNode<E>
{
    E data;
    BinaryNode left;
    BinaryNode right;
}
```

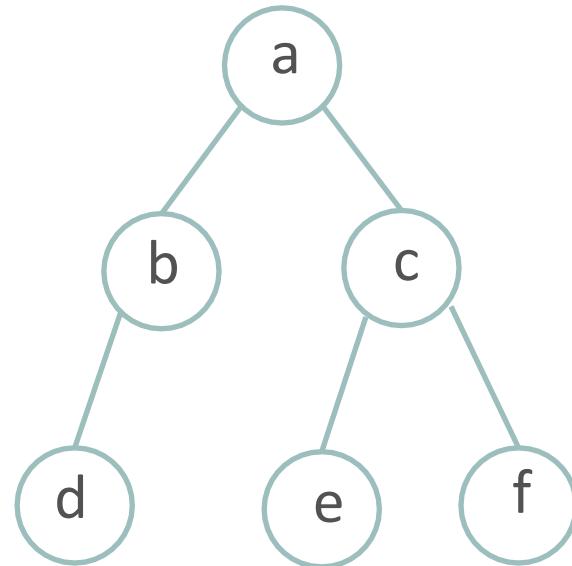
- What are the values of `left` and `right` for a leaf node?
- This is just the Node class!
  - The BinaryTree class would contain what?

```
class BinaryNode<E>
{
    E data;
    BinaryNode left;
    BinaryNode right;
}
```

```
class Tree<E> {
    BinaryNode root;
    ...
}
```

# Traversing a tree

- Traversing a *linked list* is simple
  - There is only one way to go!
- How do we traverse a binary tree if we want to visit every node?
  - E.g., we want to print out the data at every node
- How do we decide which direction to take first at each node?



# Traversal orders

- **Pre-order**
  - Use the node before traversing its children
- **In-order**
  - Traverse left child, use node, traverse right child
- **Post-order**
  - Use node after traversing both children

- **Pre-order:**

- use root

- use left subtree

- use right subtree

- **In-order:**

- use left subtree

- use root

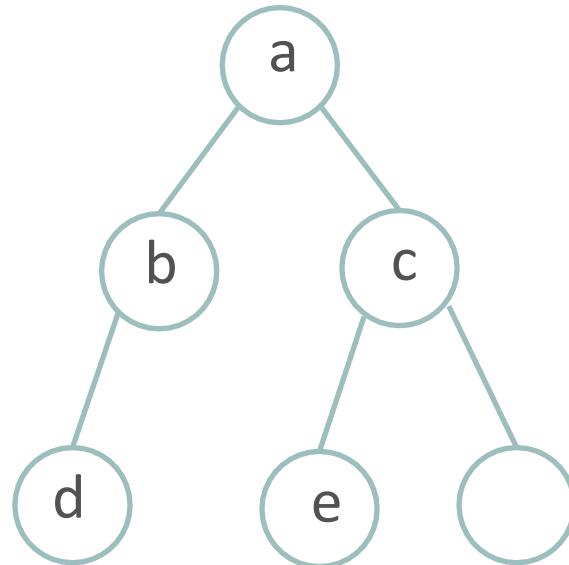
- use right subtree

- **Post-order:**

- use left subtree

- use right subtree

- use root



- **Pre-order:**

- use root

- use left subtree

- use right subtree

- **In-order:**

- use left subtree

- use root

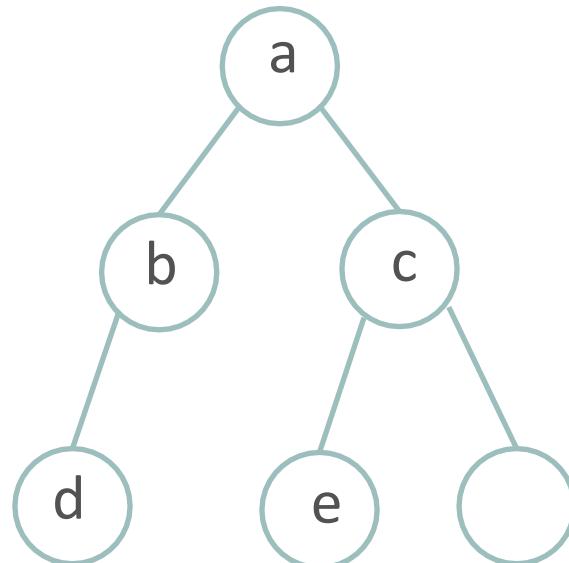
- use right subtree

- **Post-order:**

- use left subtree

- use right subtree

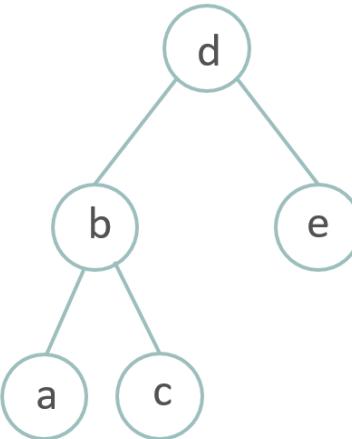
- use root



```

1 // Java program for different tree traversals
2
3 /* Class containing left and right child of current
4 node and key value*/
5 class Node
6 {
7     int key;
8     Node left, right;
9
10    public Node(int item)
11    {
12        key = item;
13        left = right = null;
14    }
15}
16
17 class BinaryTree
18 {
19     // Root of Binary Tree
20     Node root;
21
22     BinaryTree()
23     {
24         root = null;
25     }
26
27     /* Given a binary tree, print its nodes according to the
28     "bottom-up" postorder traversal. */
29     void printPostorder(Node node)
30     {
31         if (node == null)
32             return;
33
34         // first recur on left subtree
35         printPostorder(node.left);
36
37         // then recur on right subtree
38         printPostorder(node.right);
39
40         // now deal with the node
41         System.out.print(node.key + " ");
42     }
43
44     /* Given a binary tree, print its nodes in inorder*/
45     void printInorder(Node node)
46     {
47         if (node == null)
48             return;
49
50         /* first recur on left child */
51         printInorder(node.left);
52
53         /* then print the data of node */
54         System.out.print(node.key + " ");
55
56         /* now recur on right child */
57         printInorder(node.right);
58     }

```



```

59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

```

```

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(Node node)
{
    if (node == null)
        return;

    /* first print data of node */
    System.out.print(node.key + " ");

    /* then recur on left subtree */
    printPreorder(node.left);

    /* now recur on right subtree */
    printPreorder(node.right);
}

// Wrappers over above recursive functions
void printPostorder() { printPostorder(root); }
void printInorder() { printInorder(root); }
void printPreorder() { printPreorder(root); }

// Driver method
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Preorder traversal of binary tree is ");
    tree.printPreorder();

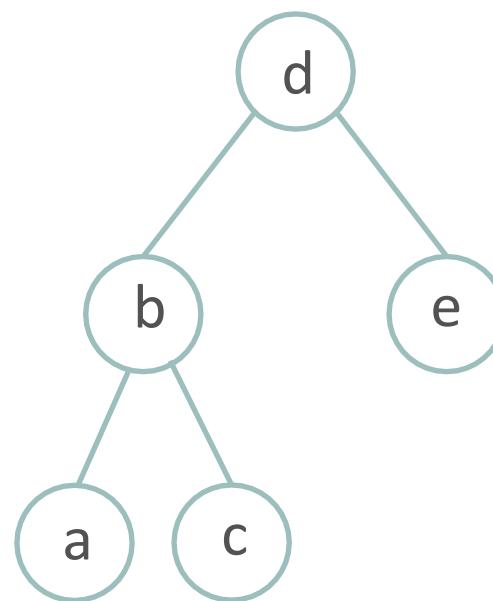
    System.out.println("\nInorder traversal of binary tree is ");
    tree.printInorder();

    System.out.println("\nPostorder traversal of binary tree is ");
    tree.printPostorder();
}

```

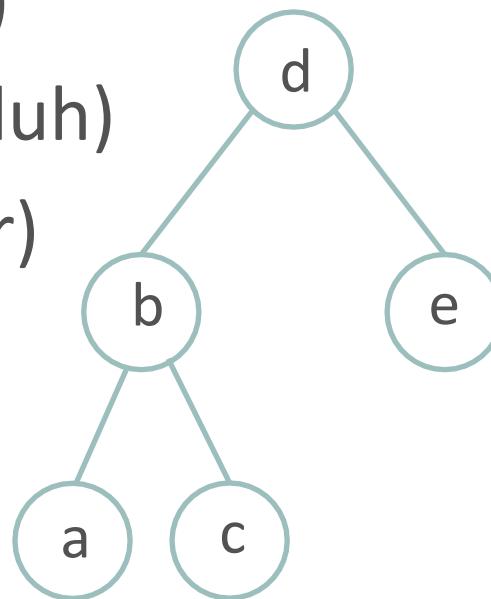
- What order are the nodes printed using pre-order traversal?

- A) d b a c e
- B) a b c d e
- C) a c b e d



- What order are the nodes printed using pre-order traversal?

- A) d b a c e (pre-order)
- B) a b c d e (in-order, duh)
- C) a c b e d (post-order)



```
Public void printPreorder(BinaryNode root)
{
    System.out.print(" " + root.data);
    printPreorder(root.left);
    printPreorder(root.right);
}
```

Are we missing anything?

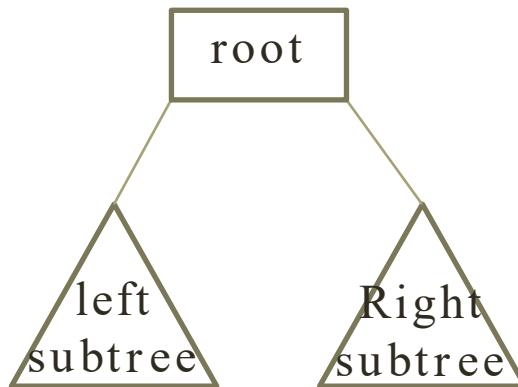
```
Public void printPreorder(BinaryNode root)
{
    if (root != null)
    {
        System.out.print(" " + root.data);
        printPreorder(root.left);
        printPreorder(root.right);
    }
}
```

# How to compute the height of a binary tree?

```
int height (BinaryNode n)
```

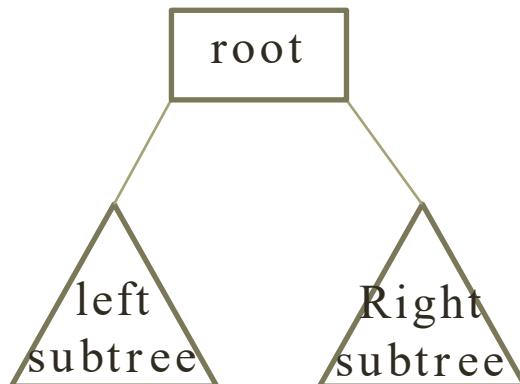
```
{
```

```
}
```



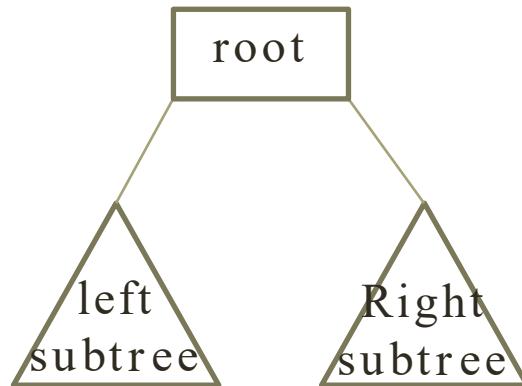
# How to compute the height of a binary tree?

```
int height (BinaryNode n)
{
    return max (height (n.left) ,
                height (n.right) ) + 1;
}
```



# How to compute the height of a binary tree?

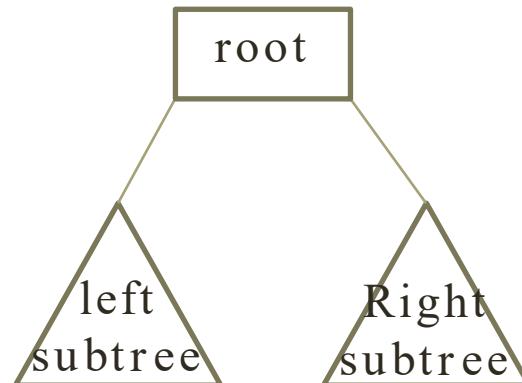
```
int height (BinaryNode n)
{
    return max (height (n.left) ,
                height (n.right) ) + 1;
}
```



Are we missing  
anything?

# How to compute the height of a binary tree?

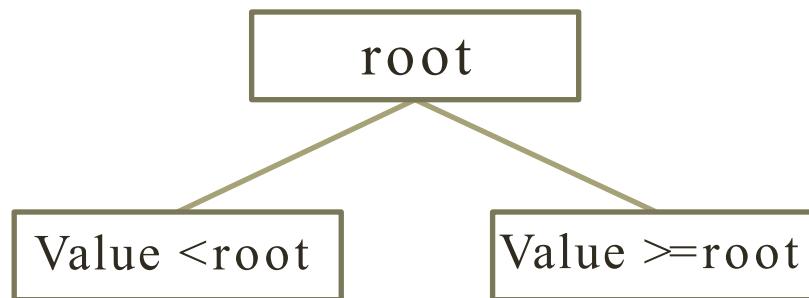
```
int height(BinaryNode n)
{
    if (n == null)
        return 0;
    else
        return max(height(n.left),
                   height(n.right)) + 1;
}
```



# Binary Search Trees...

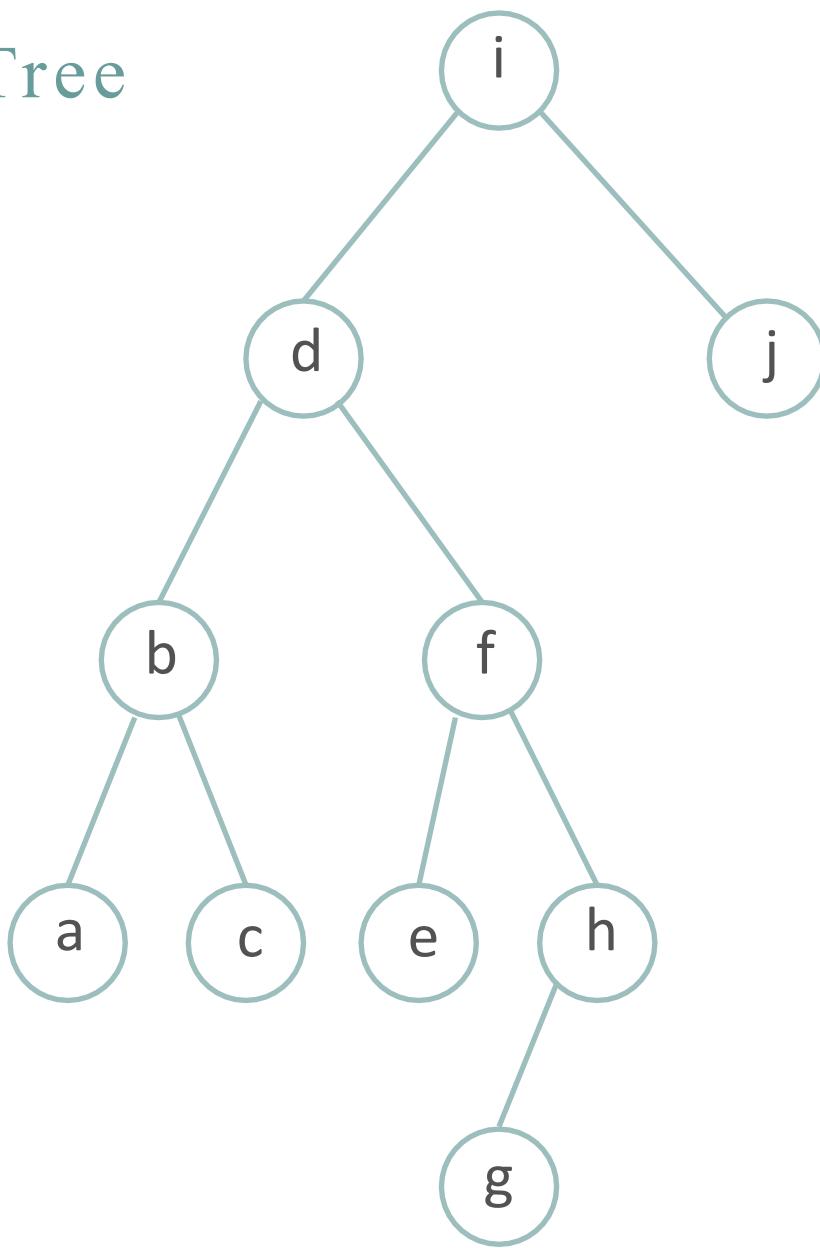
- A **binary search tree** is a binary tree with a restriction on the **ordering** of nodes
  - All items in the **left** subtree of a node are *less than* the item in the node
  - All items in the **right** subtree of a node are *greater than or equal to* the item in the node

- A **binary search tree** is a binary tree with a restriction on the **ordering** of nodes
  - All items in the **left** subtree of a node are *less than* the item in the node
  - All items in the **right** subtree of a node are *greater than or equal to* the item in the node



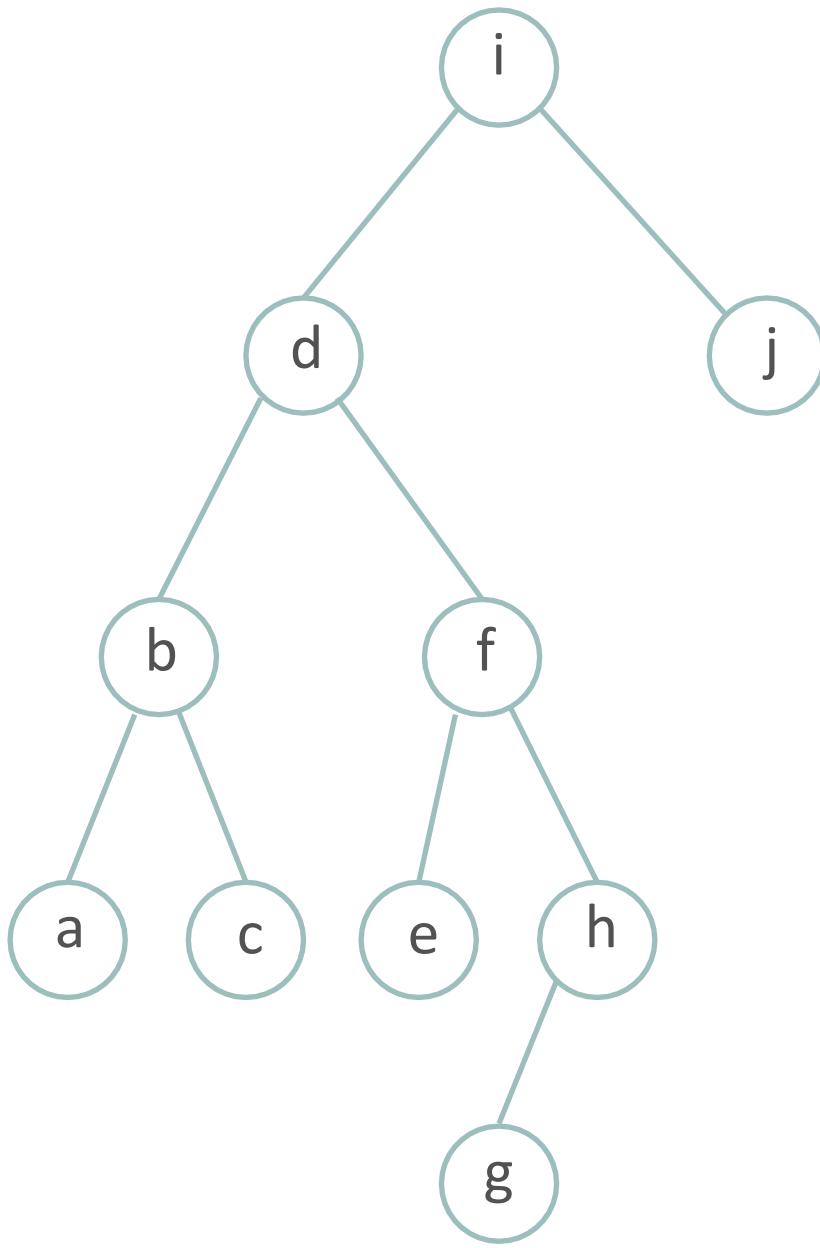
BSTs allow for fast searching of nodes

# Binary Search Tree

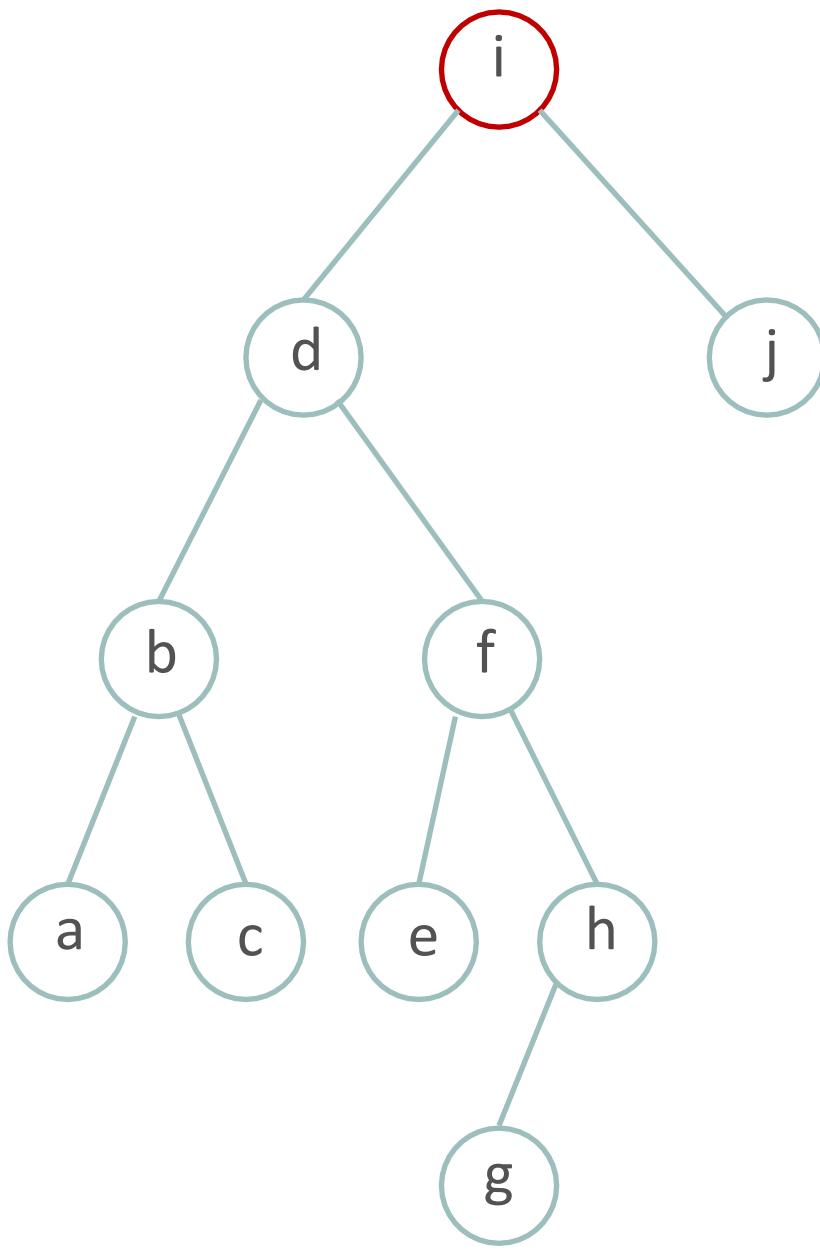


# Search

looking for **h**

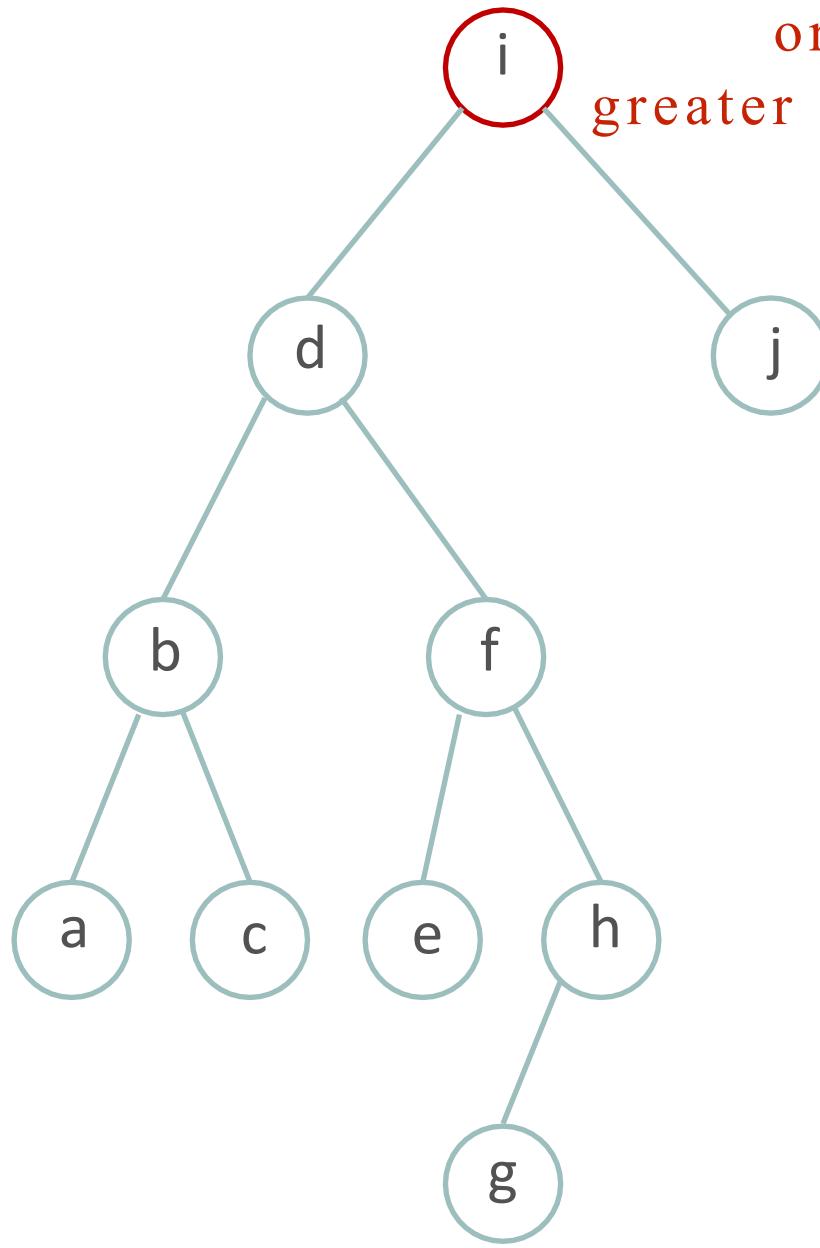


looking for **h**

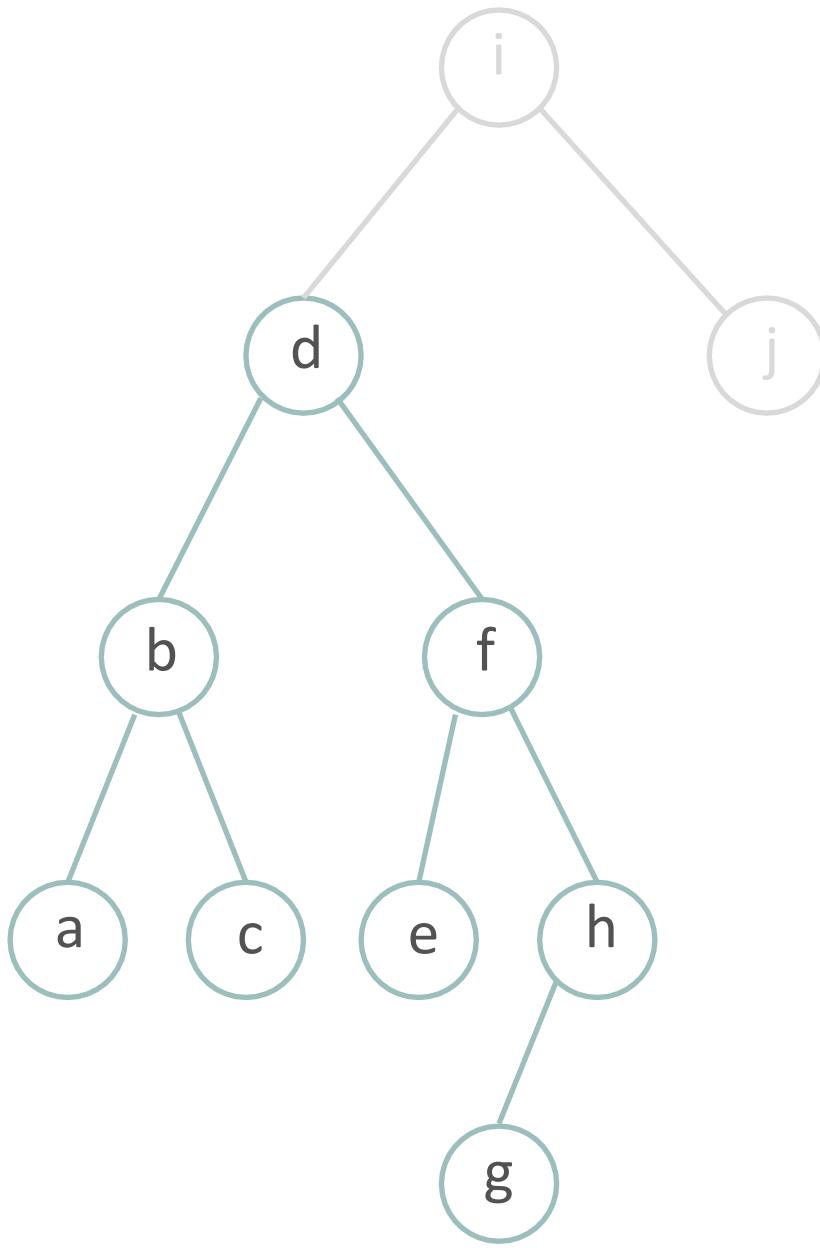


is **h** less than  
or  
greater than **i**?

looking for **h**

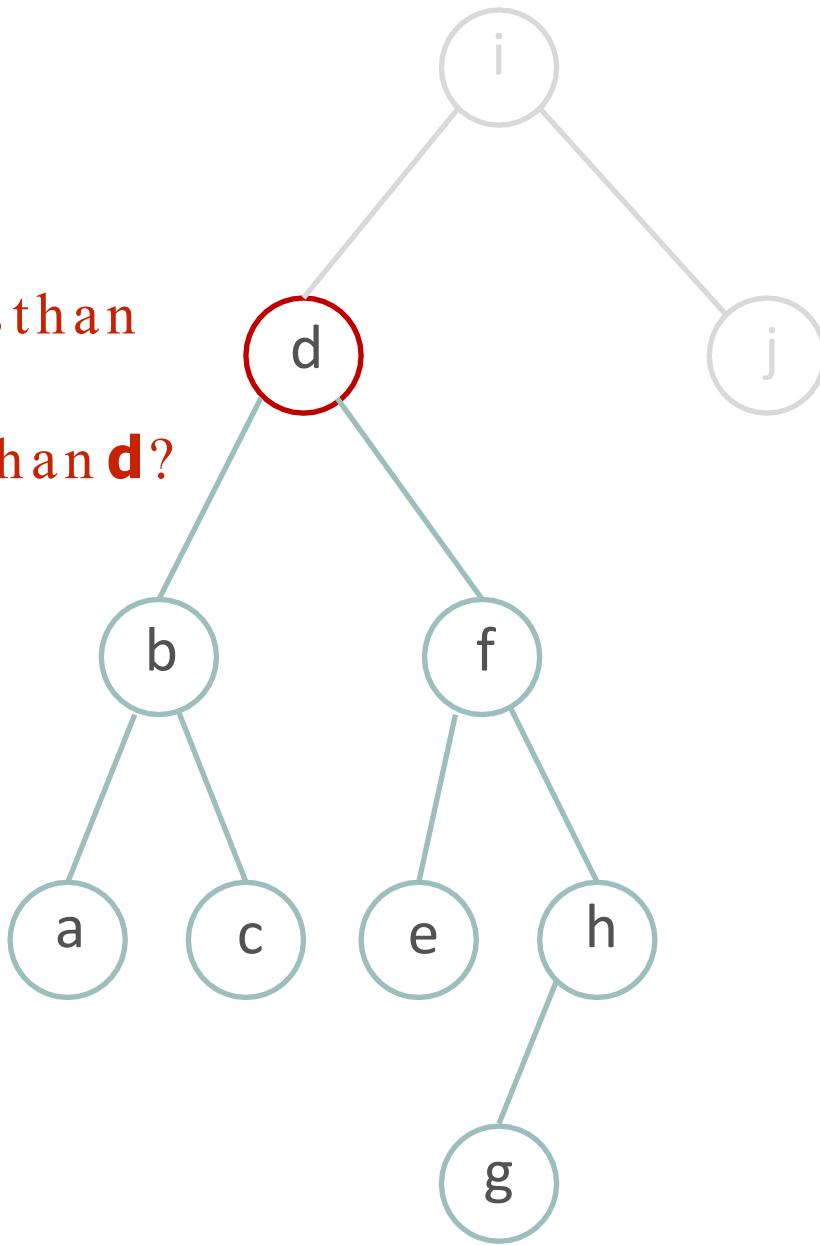


looking for **h**

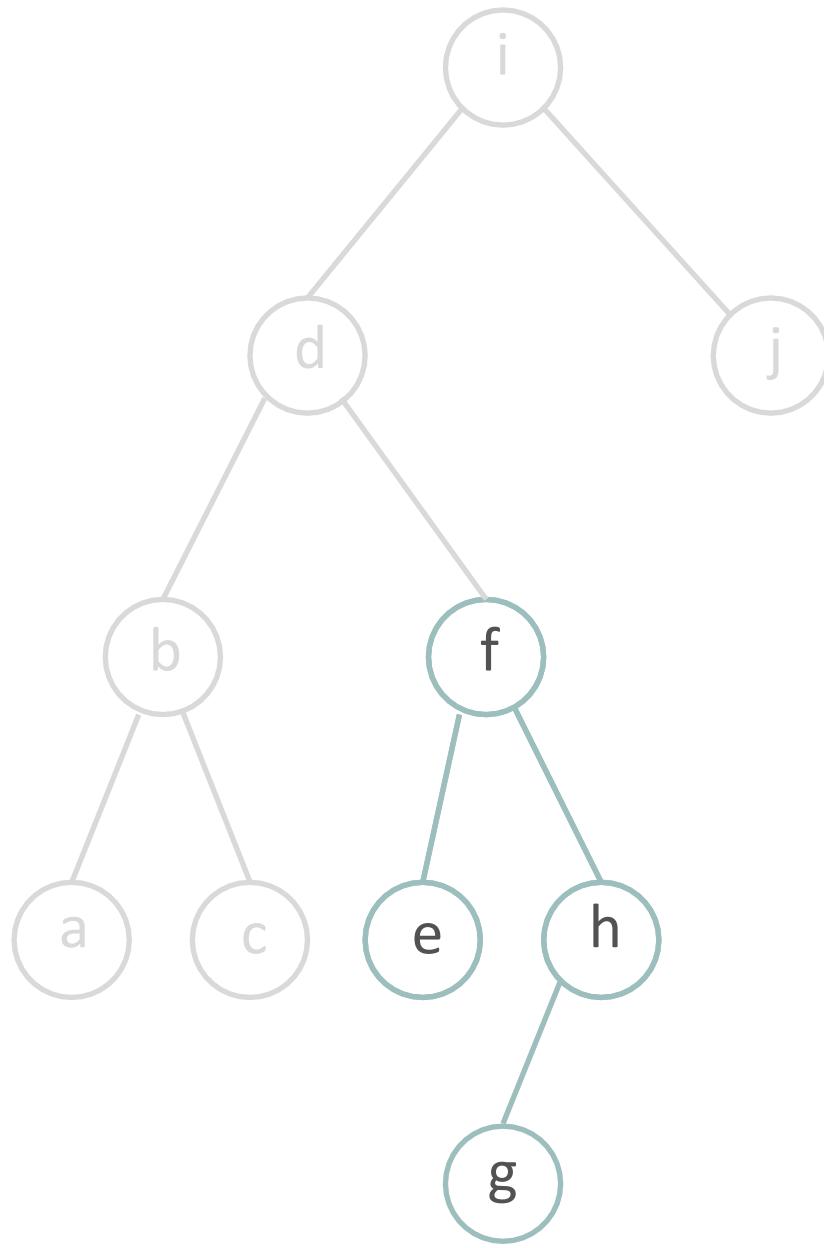


looking for **h**

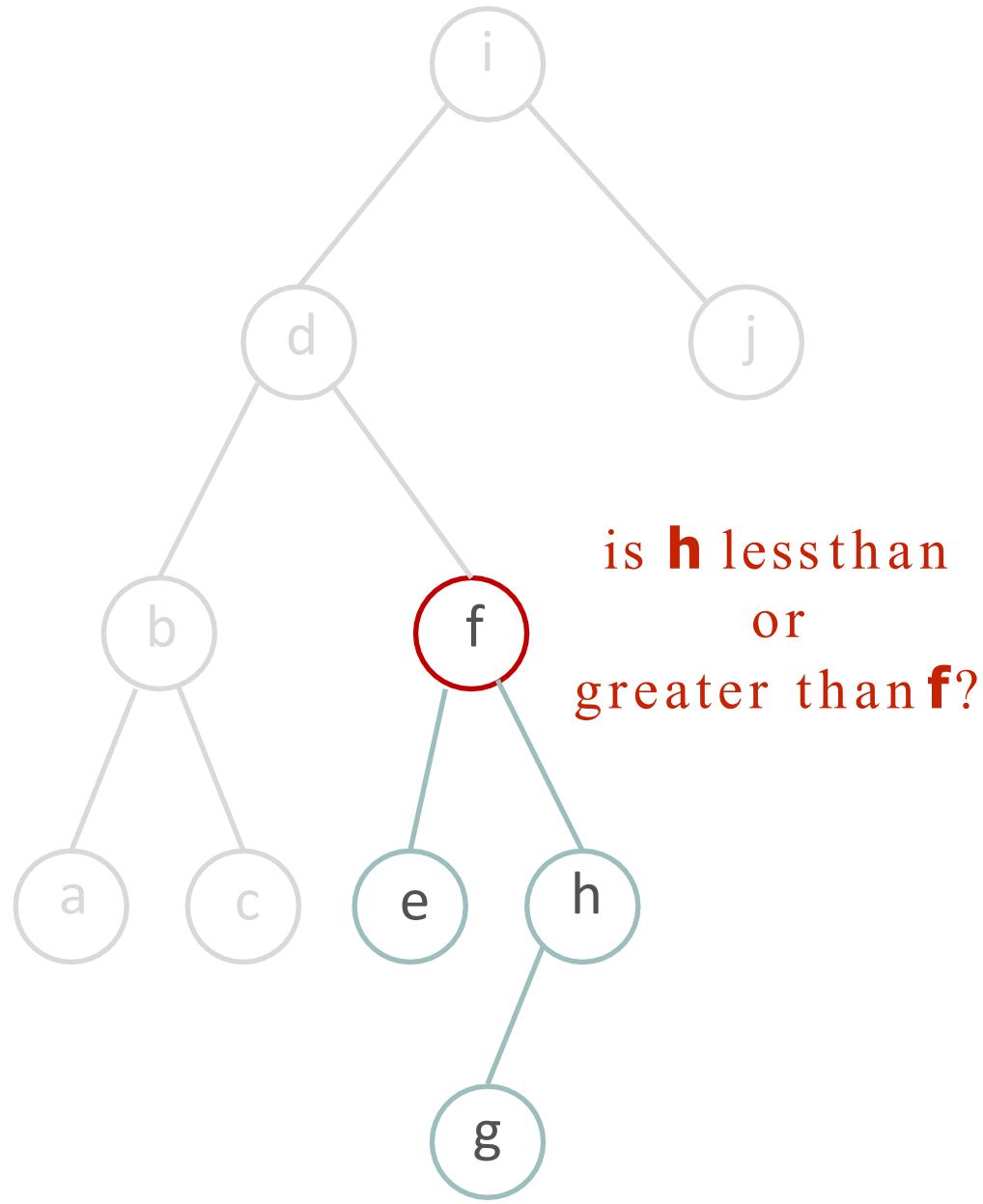
is **h** less than  
or  
greater than **d**?



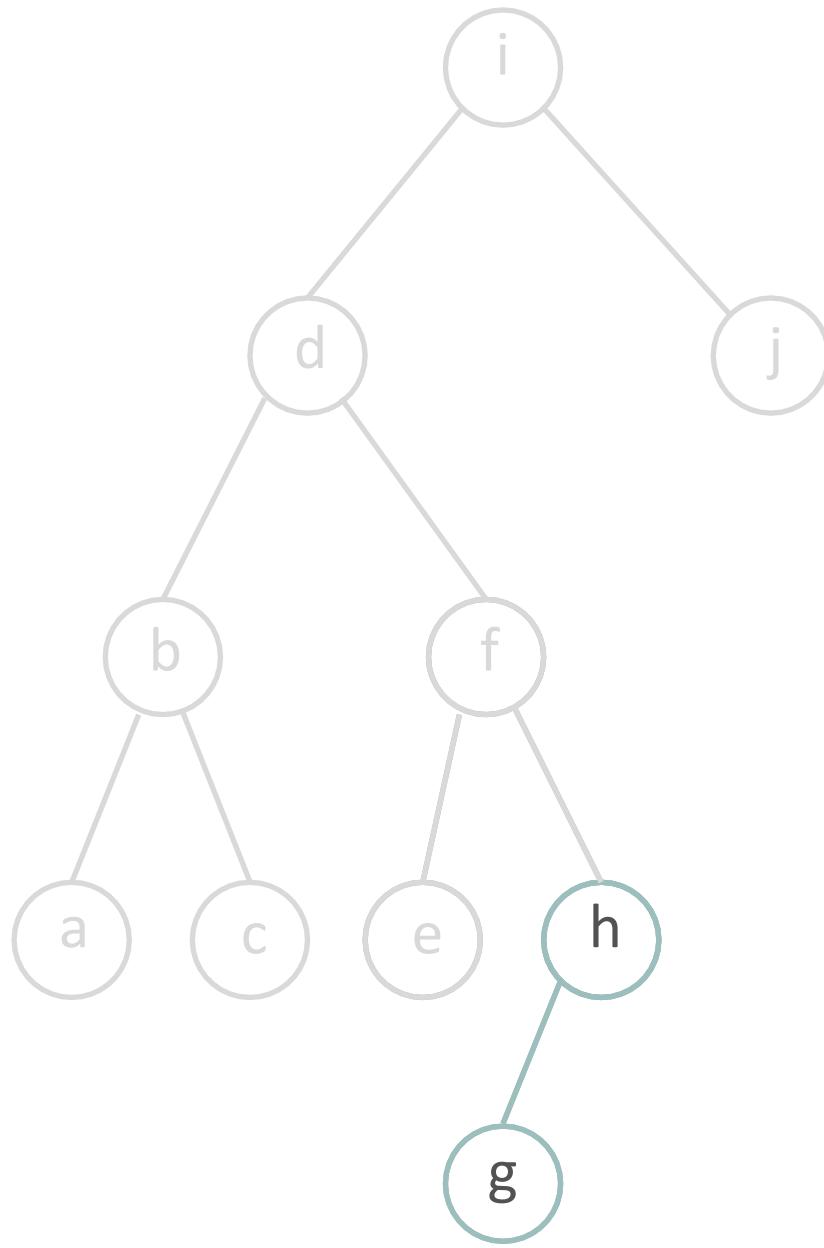
looking for **h**



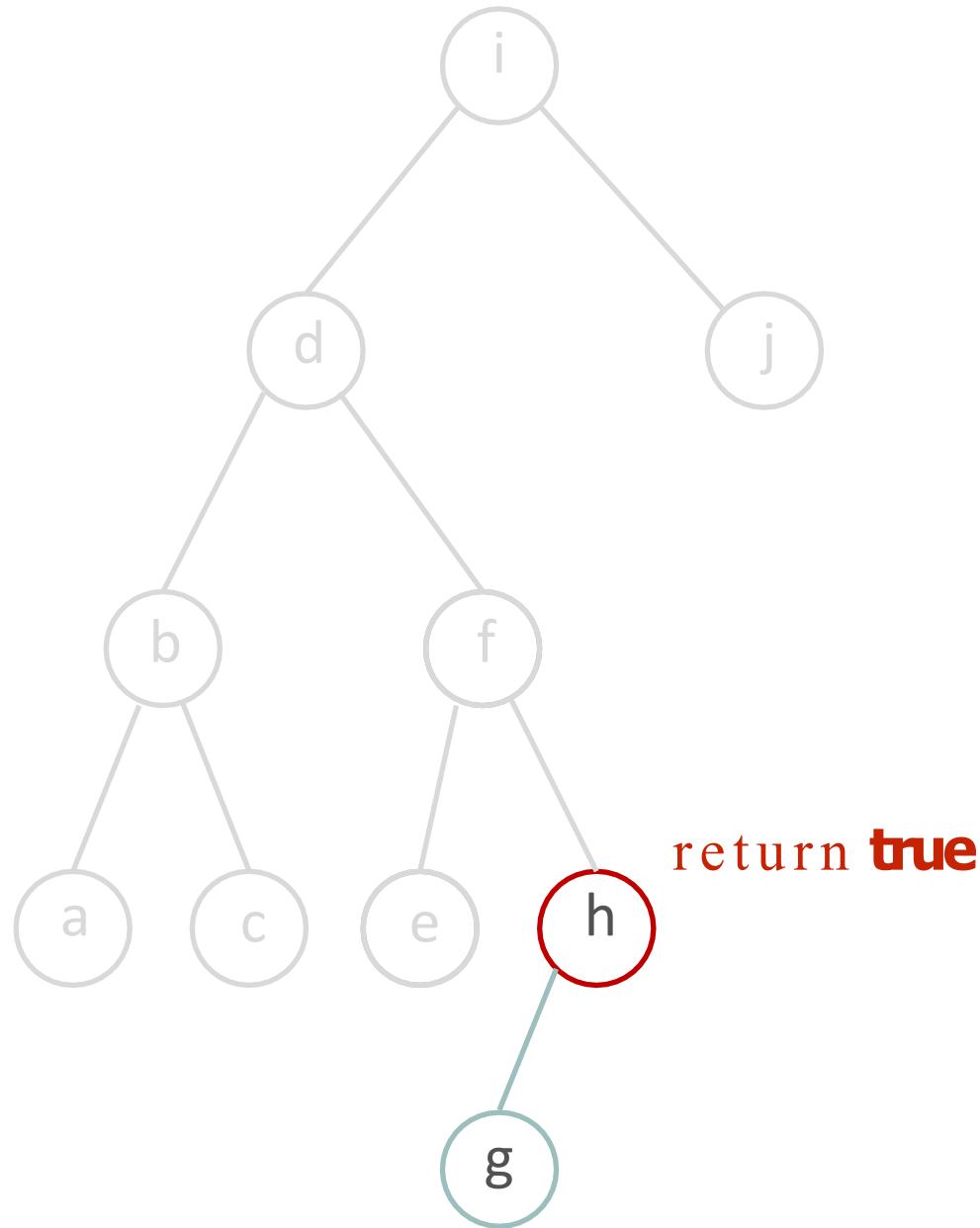
looking for **h**



looking for **h**



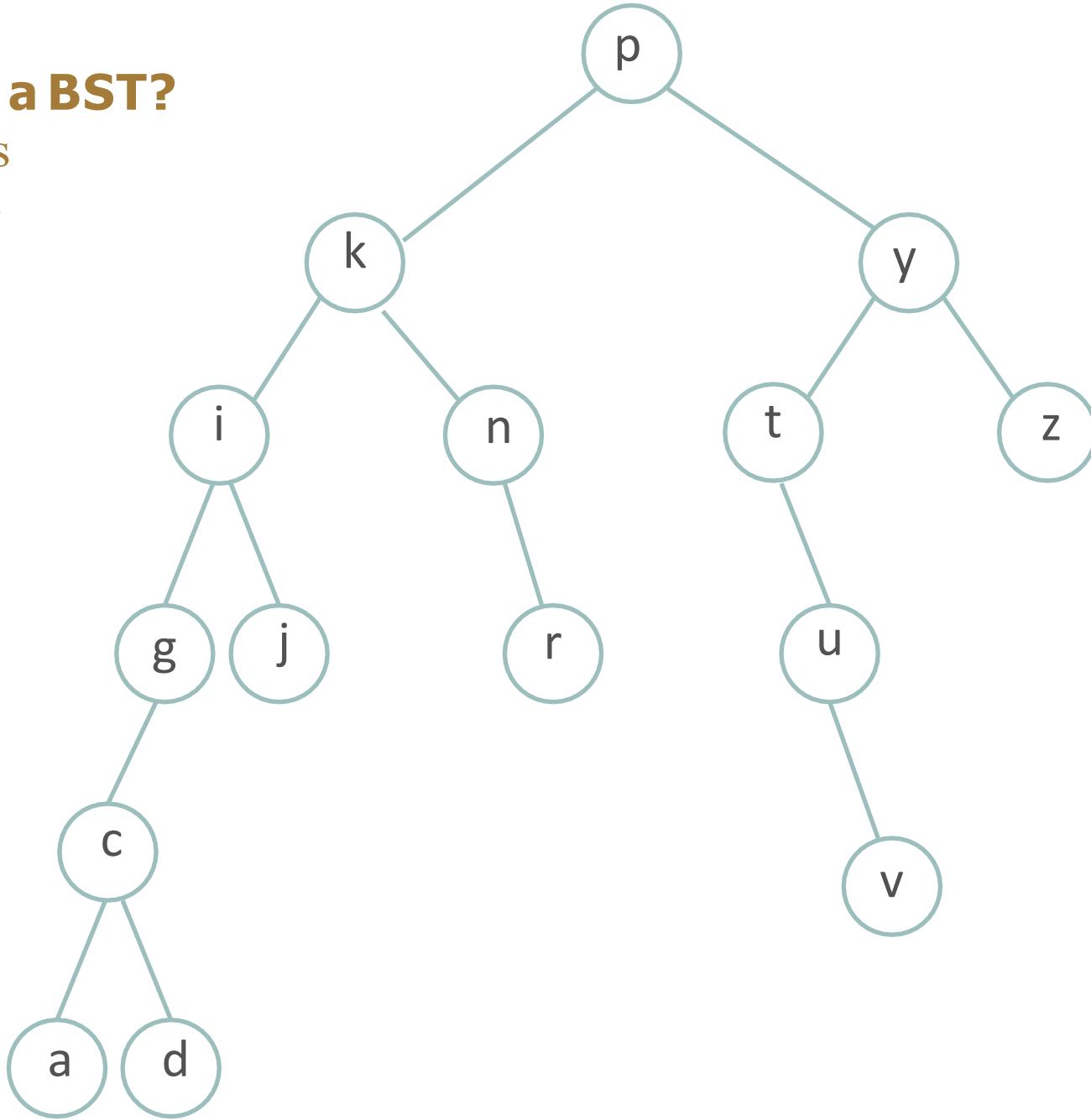
looking for **h**



- NOTE: at each step we eliminate approximately half of the tree
  - What does this imply about the search complexity?
- What if the item is not found in the tree?
  - How do we know when to stop and return `false`?

**Is this a BST?**

- A) yes
- B) no



Next time...

- Tree-continued
- Start your assignment early



# Trees

## CSC220|Computer Programming 2

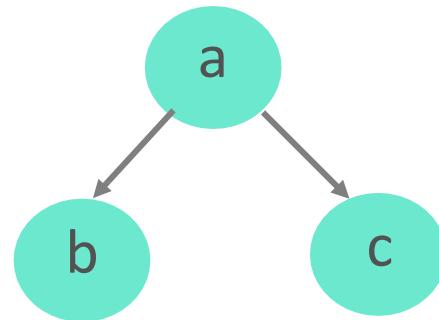
Last Time...

# Trees

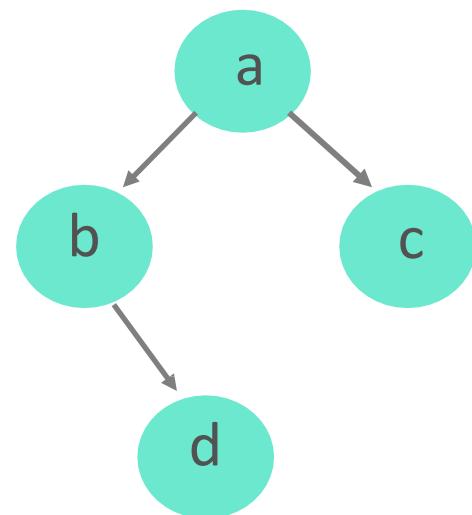
# Tree

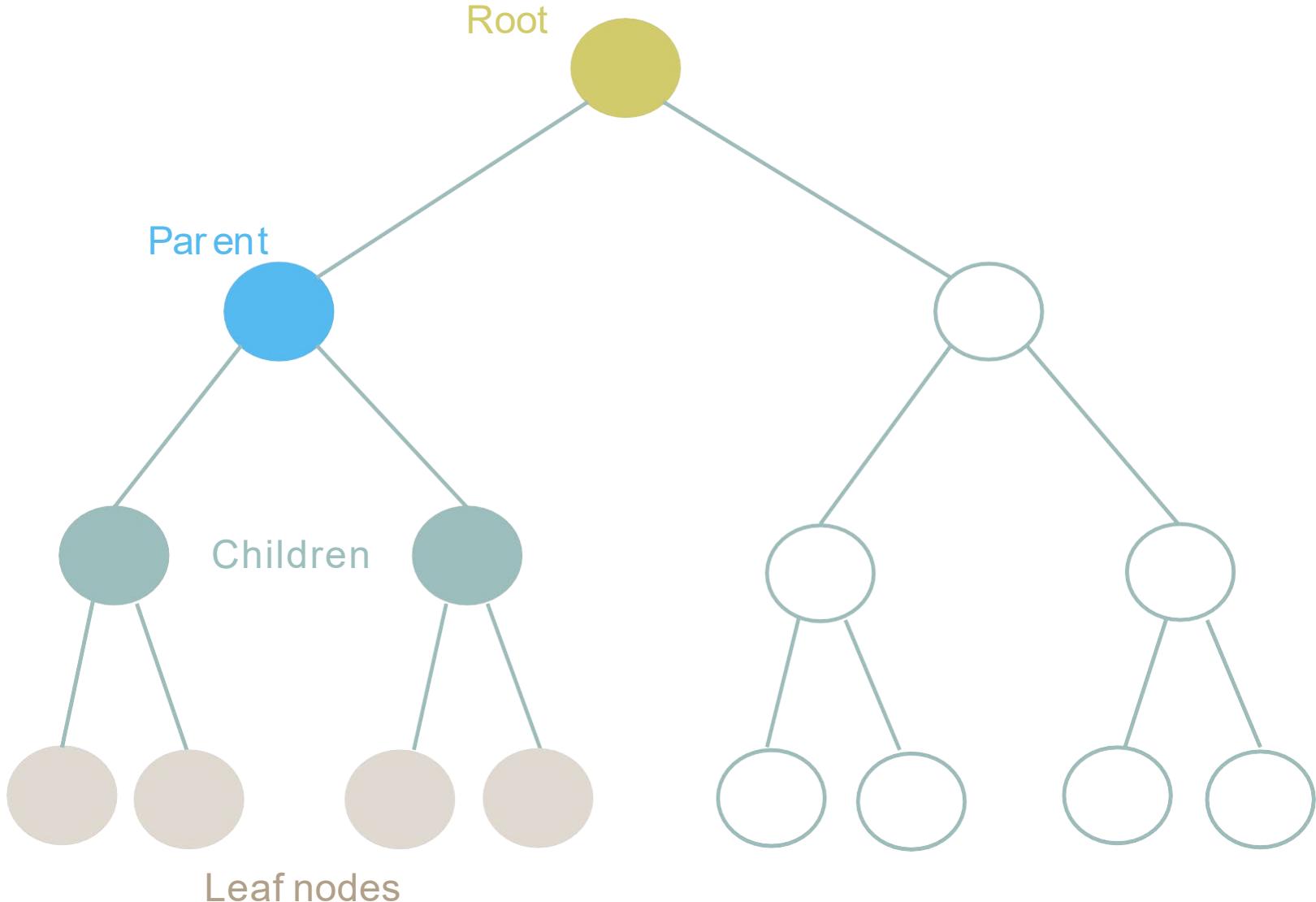
- A linked data structure with a hierarchical formation
- Trees can have multiple links, called branches

There are multiple directions  
you can take at any  
given node



- There is a strict parent-to-child relationship among nodes
  - Links *only* go from parent to child
    - *Not from child to parent*
    - *Not from sibling to sibling*
- Every node has exactly **one** parent, except for the **root**, which has none
- There is exactly one path from the root to any other node





# Binary Trees

- **Binary trees** are a special case of a tree in which a node can have **AT MOST** two children
- These nodes are designated *left* and *right*
- In this class we will mostly concentrate on binary trees

What should the implementation of a  
binary tree look like?  
What about a binary tree node?

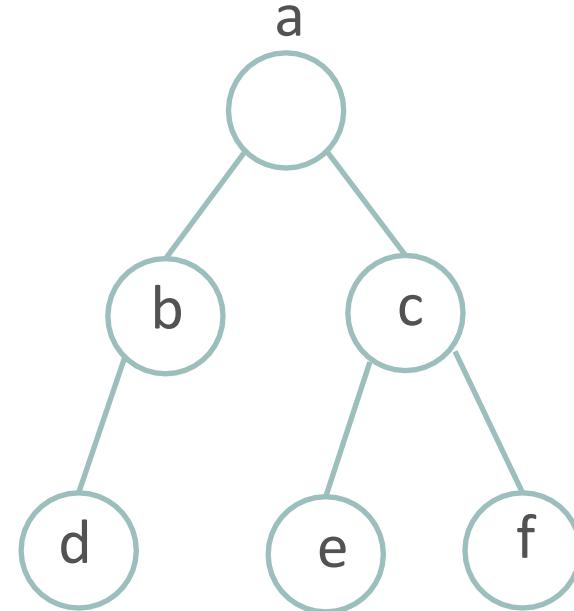
# Traversing a tree (redo)

- Traversing a *linked list* is simple (ok in theory it's tricky, but after today -> easy)

- How do we traverse a binary tree if we want to visit every node?

- E.g., we want to print out the data at every node

- How do we decide which direction to take first at each node?



# Traversal orders

- **Pre-order**
  - Use the node before traversing its children
  - Root, <left subtree>, <right subtree>
- **In-order**
  - Traverse left child, use node, traverse right child
  - <left subtree>, root, <right subtree>
- **Post-order**
  - Use node after traversing both children
  - <left subtree>, <right subtree>, root

- **Pre-order:**

- use root

- use left subtree

- use right subtree

- **In-order:**

- use left subtree

- use root

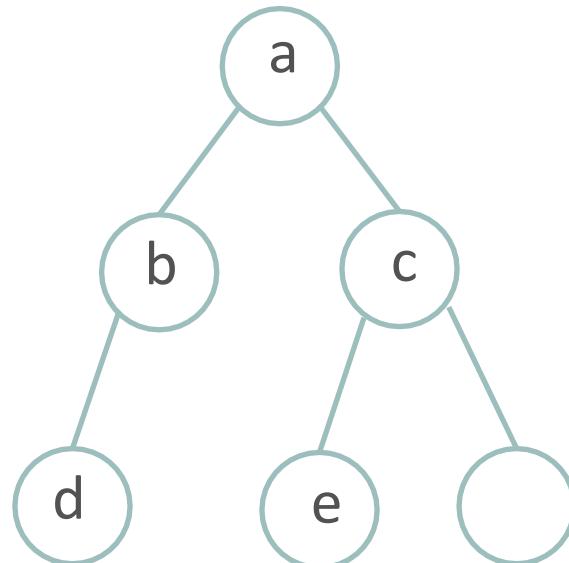
- use right subtree

- **Post-order:**

- use left subtree

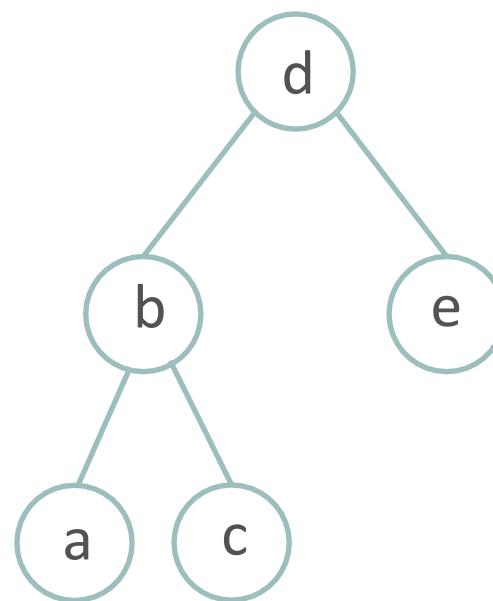
- use right subtree

- use root



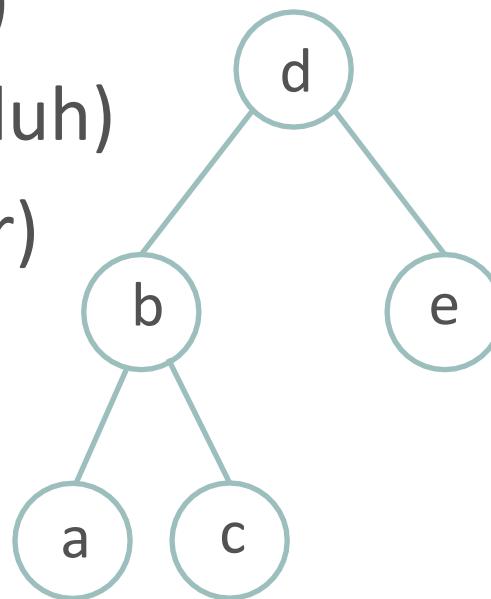
- What order are the nodes printed using pre-order traversal?

- A) d b a c e
- B) a b c d e
- C) a c b e d



- What order are the nodes printed using pre-order traversal?

- A) d b a c e (pre-order)
- B) a b c d e (in-order, duh)
- C) a c b e d (post-order)



- Example on the board...
- Non technical way:
  - Pre-order – “go with the flow”
  - Post-order – “last time visited”
  - In-order – “scan tree left to right”

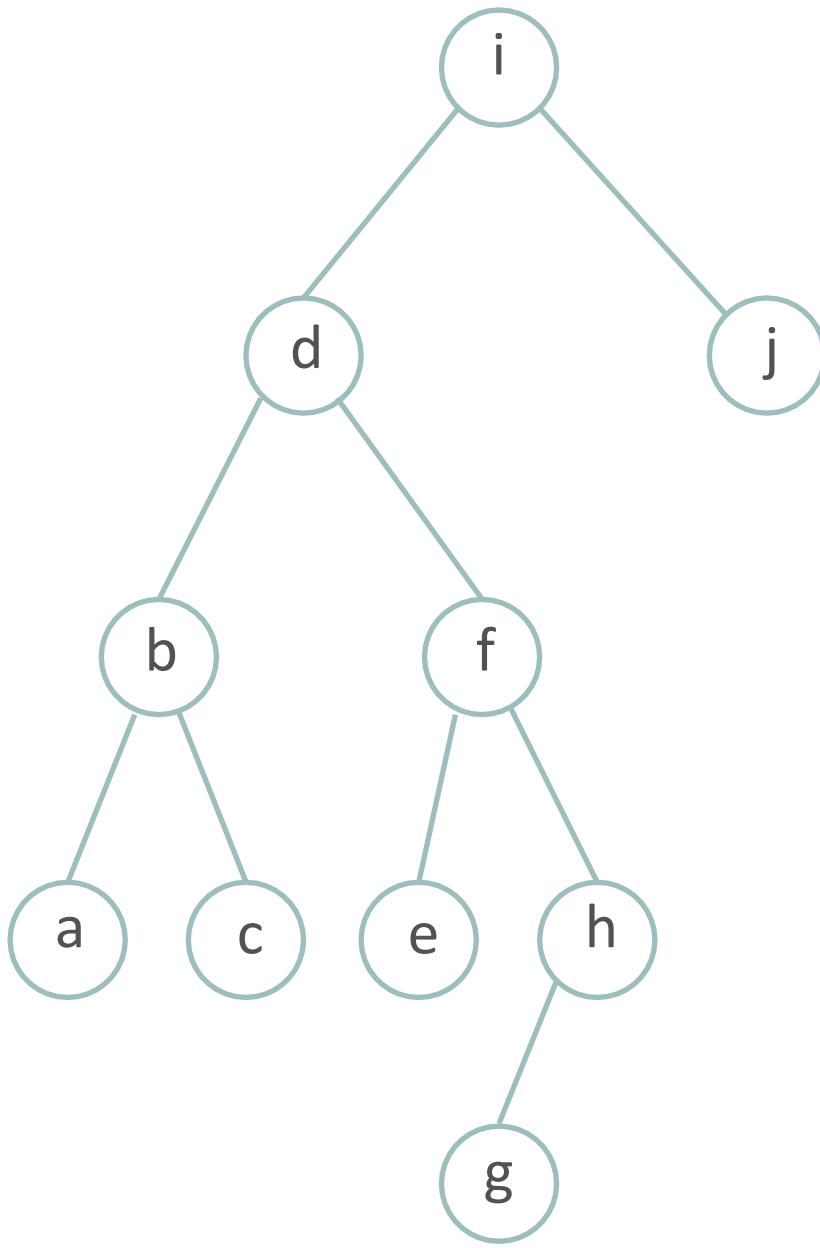
```
Public void printPreorder(BinaryNode root)
{
    System.out.print(" " + root.data);
    printPreorder(root.left);
    printPreorder(root.right);
}
```

Are we missing anything?

```
Public void printPreorder(BinaryNode root)
{
    if (root != null)
    {
        System.out.print(" " + root.data);
        printPreorder(root.left);
        printPreorder(root.right);
    }
}
```

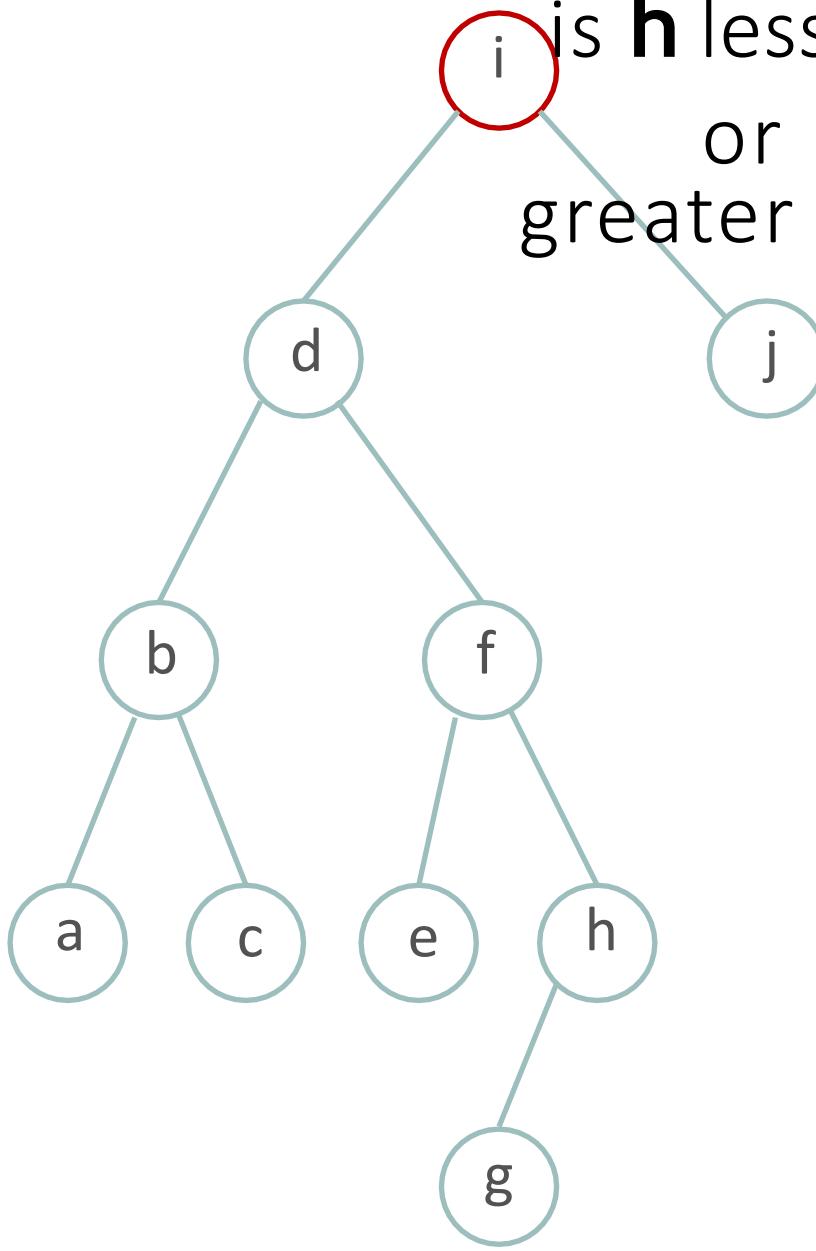
# Search

looking for h

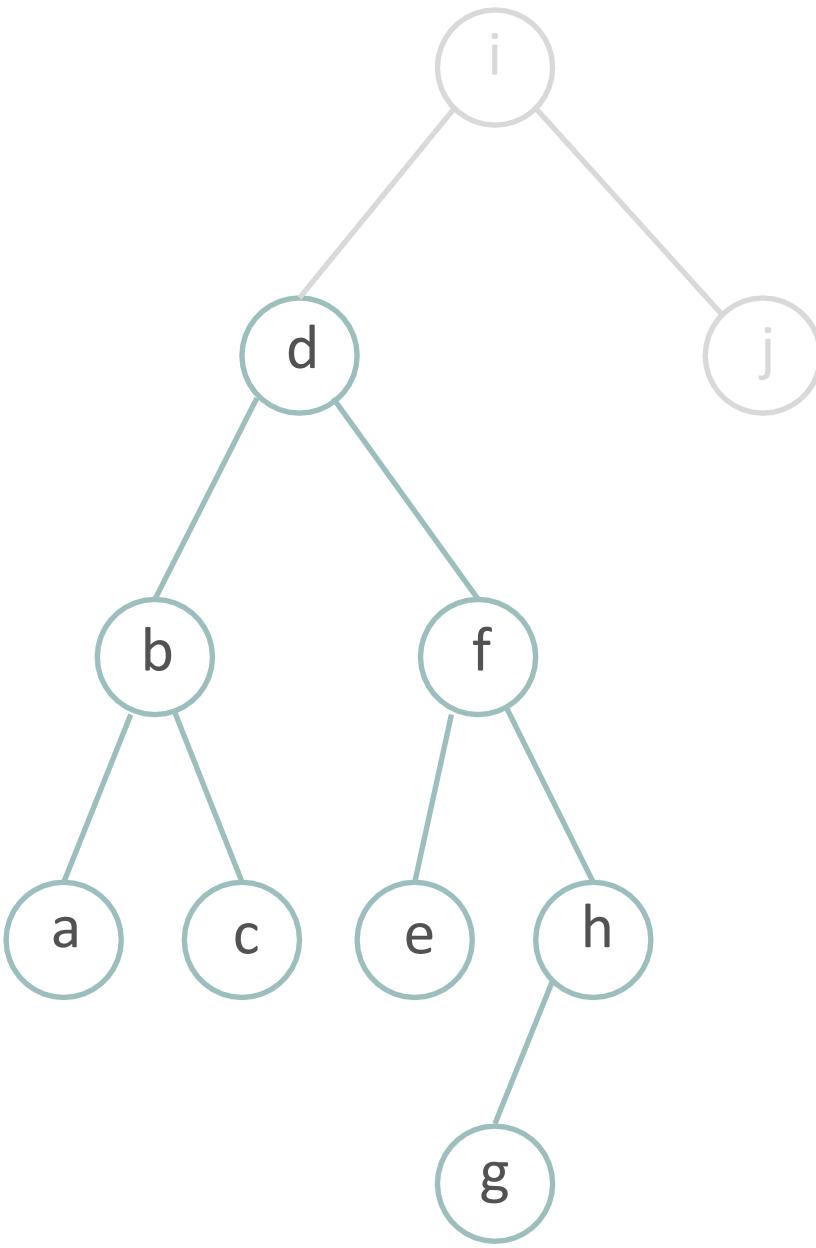


looking for h

i is h less than  
or  
greater than i?

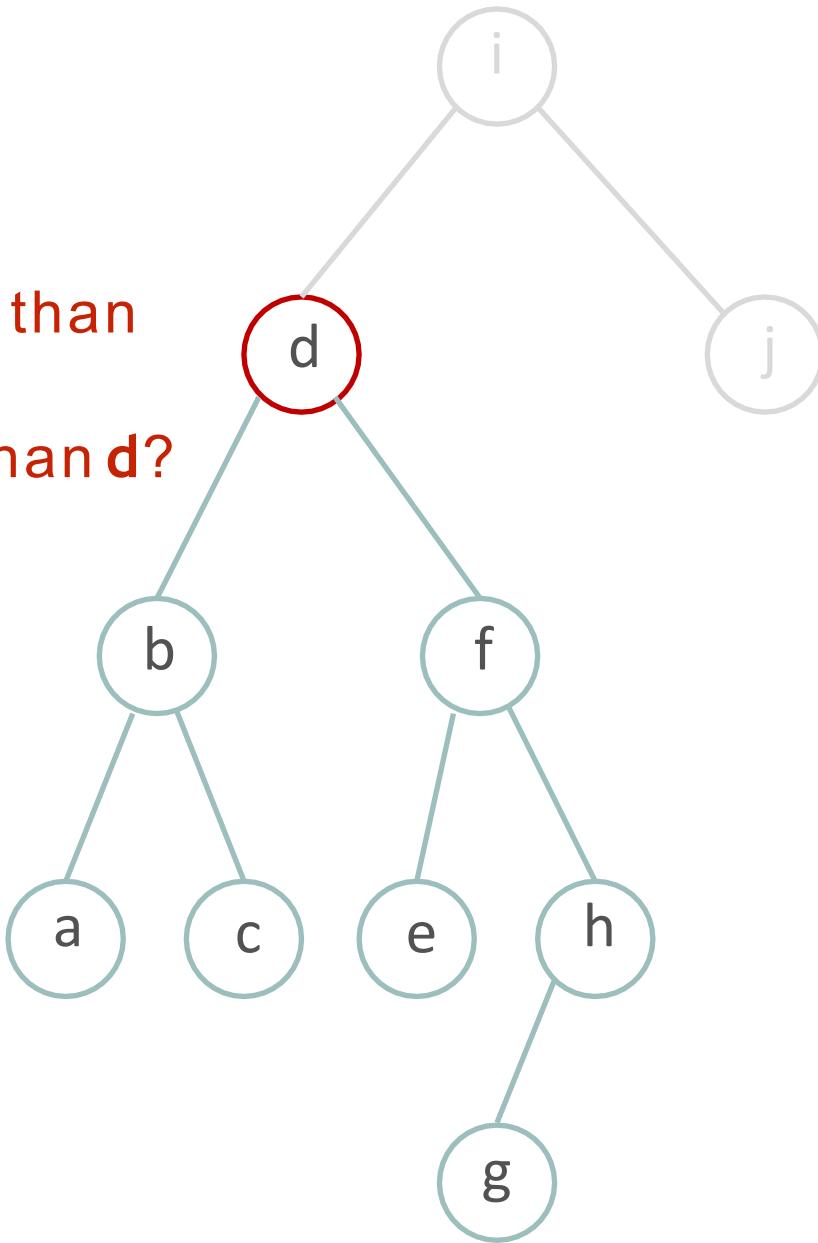


looking for h

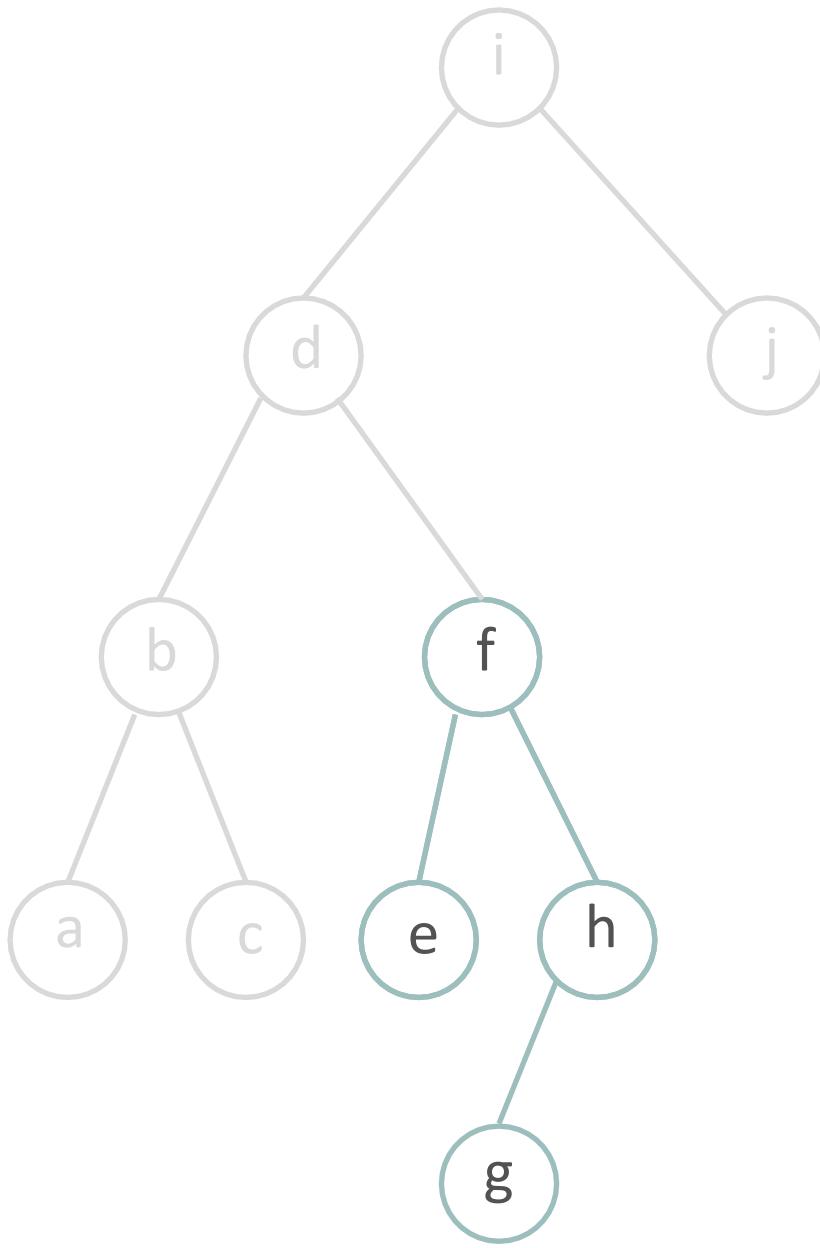


looking for **h**

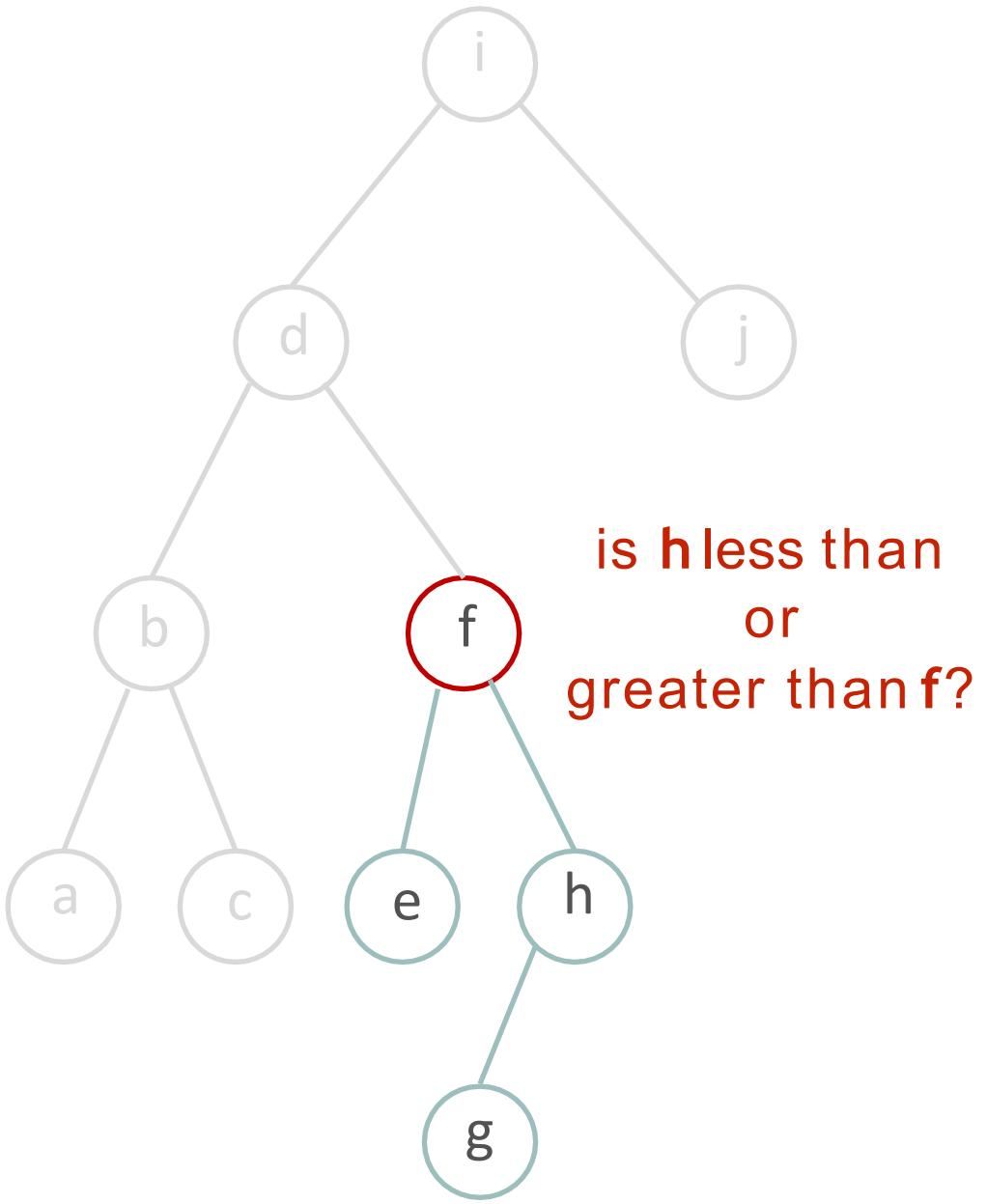
is **h** less than  
or  
greater than **d**?



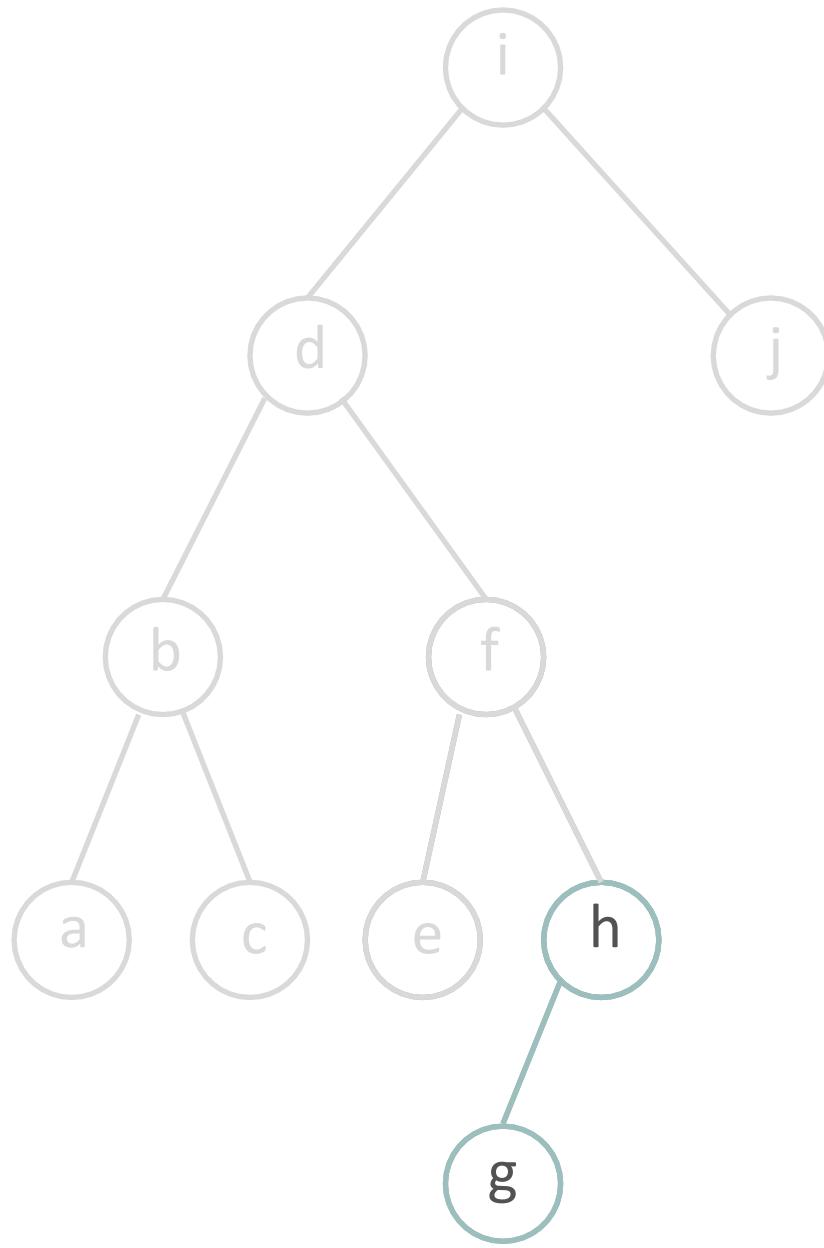
looking for h



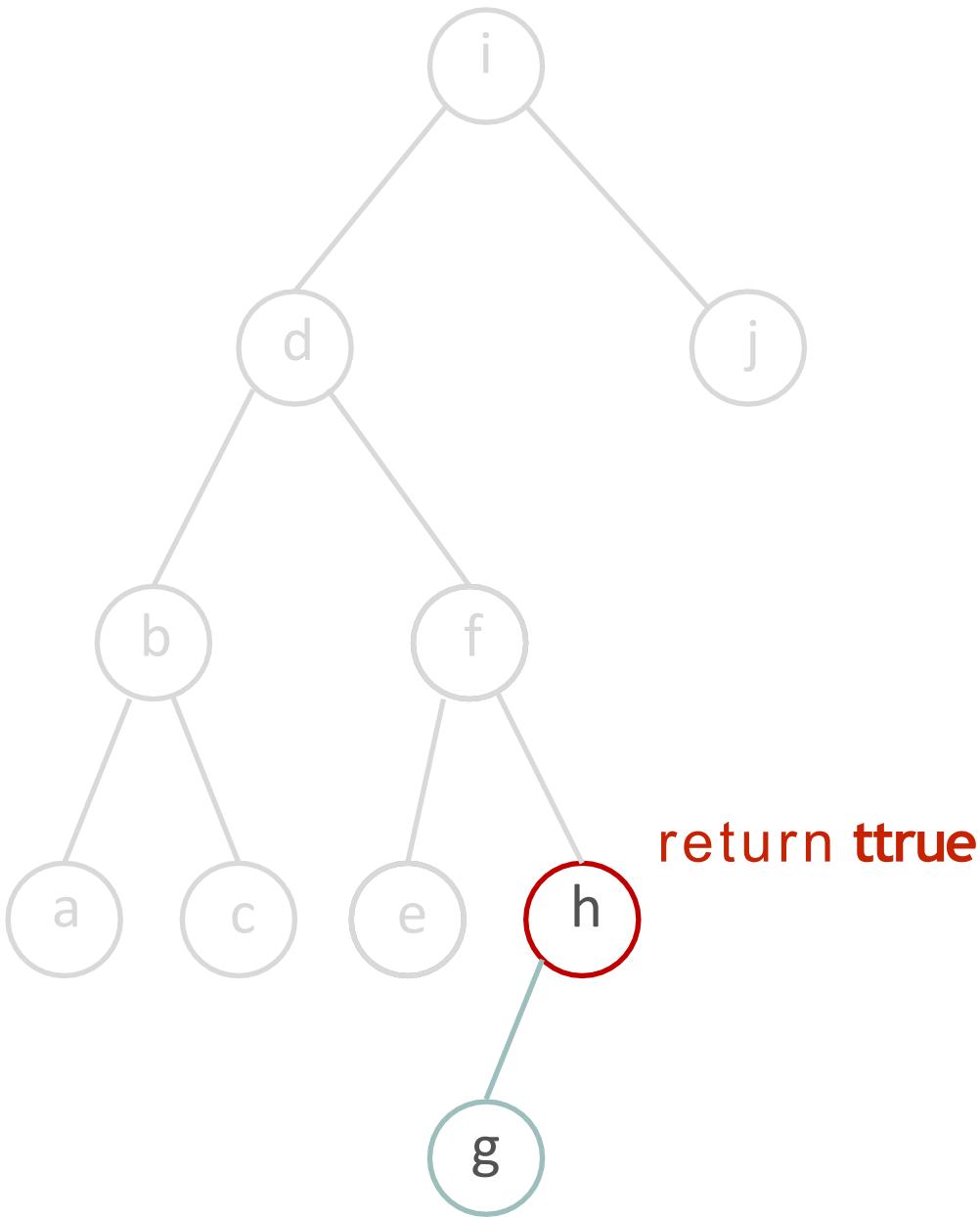
looking for **h**



looking for h



looking for **h**



- NOTE: at each step we eliminate approximately half of the tree
  - What does this imply about the search complexity?
- What if the item is not found in the tree?
  - How do we know when to stop and return `false`?

# How does this look like?

```
public boolean search(String s)
{
    System.out.println("searching for " + s);
    return searchRecursive(s, root);
}
```

# How does this look like?

```
public boolean search(String s)
{
    System.out.println("searching for " + s);
    return searchRecursive(s, root);
```



Helper method

# How does this look like?

```
public boolean search(String s)
{
    System.out.println("searching for " + s);
    return searchRecursive(s, root);
```

Helper method

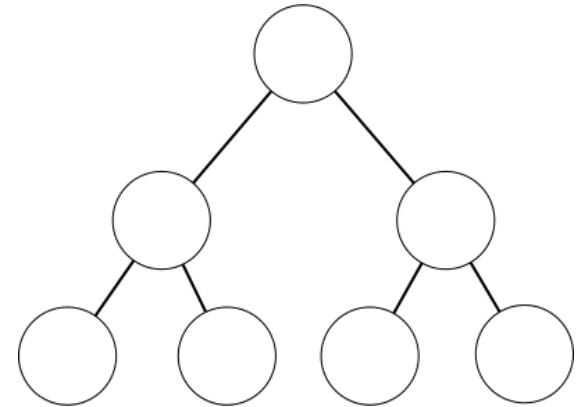
---

```
private boolean searchRecursive(String s, BSTNode n)
{

    // Reached the bottom of the tree
    if(n == null)
        return false;

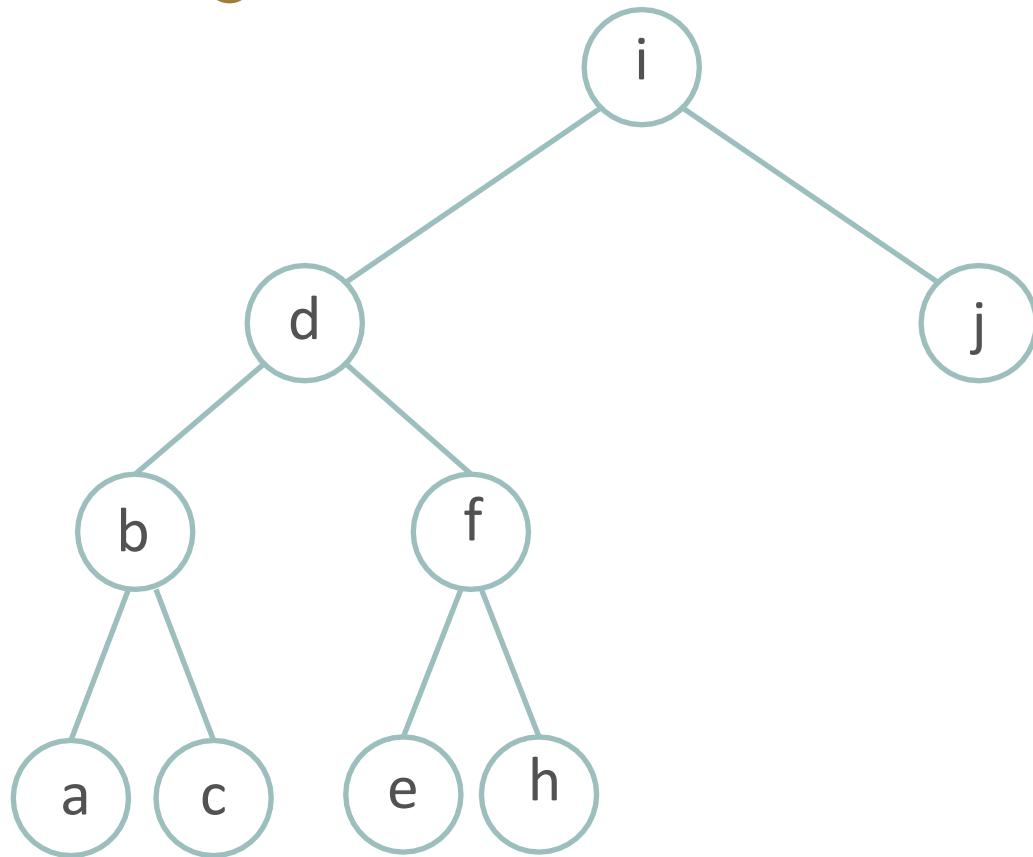
    // Found the item at this node
    if(s.equals(n.data))
        return true;

    // Otherwise, search in left or right subtree
    if(s.compareTo(n.data) < 0)
    {
        System.out.println("\t at node " + n + ", going left");
        return searchRecursive(s, n.left);
    }
    else
    {
        System.out.println("\t at node " + n + ", going right");
        return searchRecursive(s, n.right);
    }
}
```



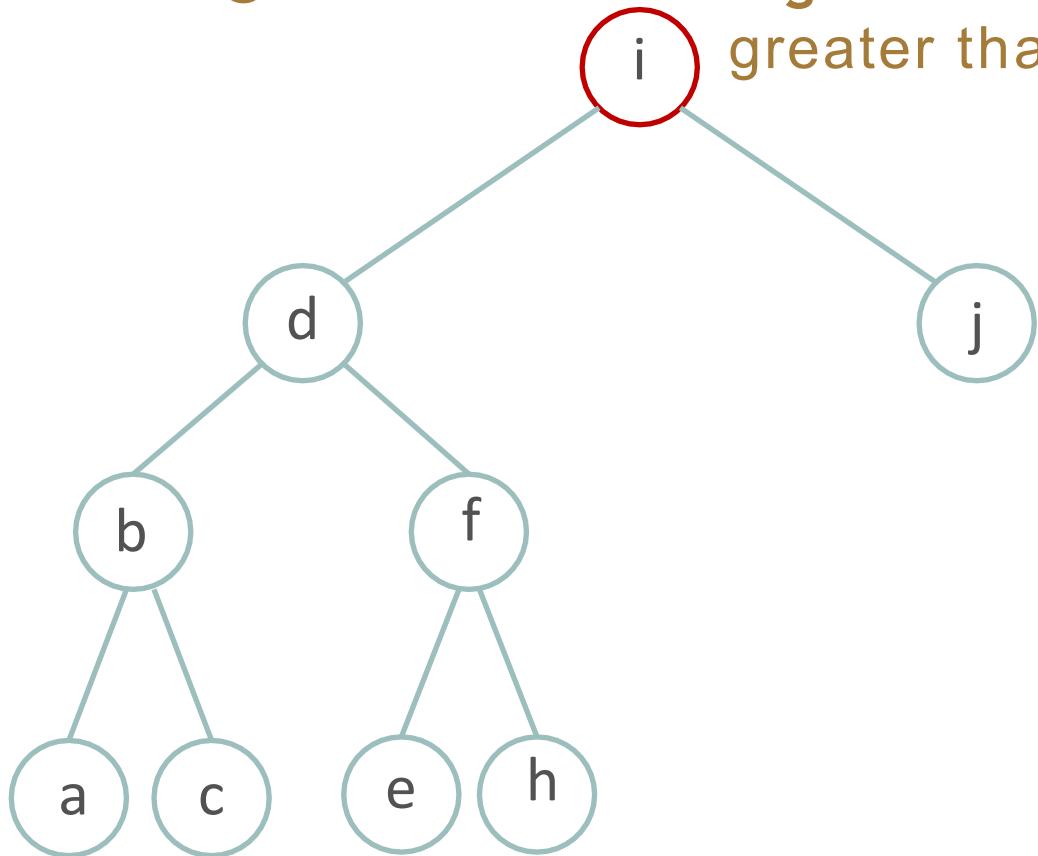
# Insertion

We want to insert **g**

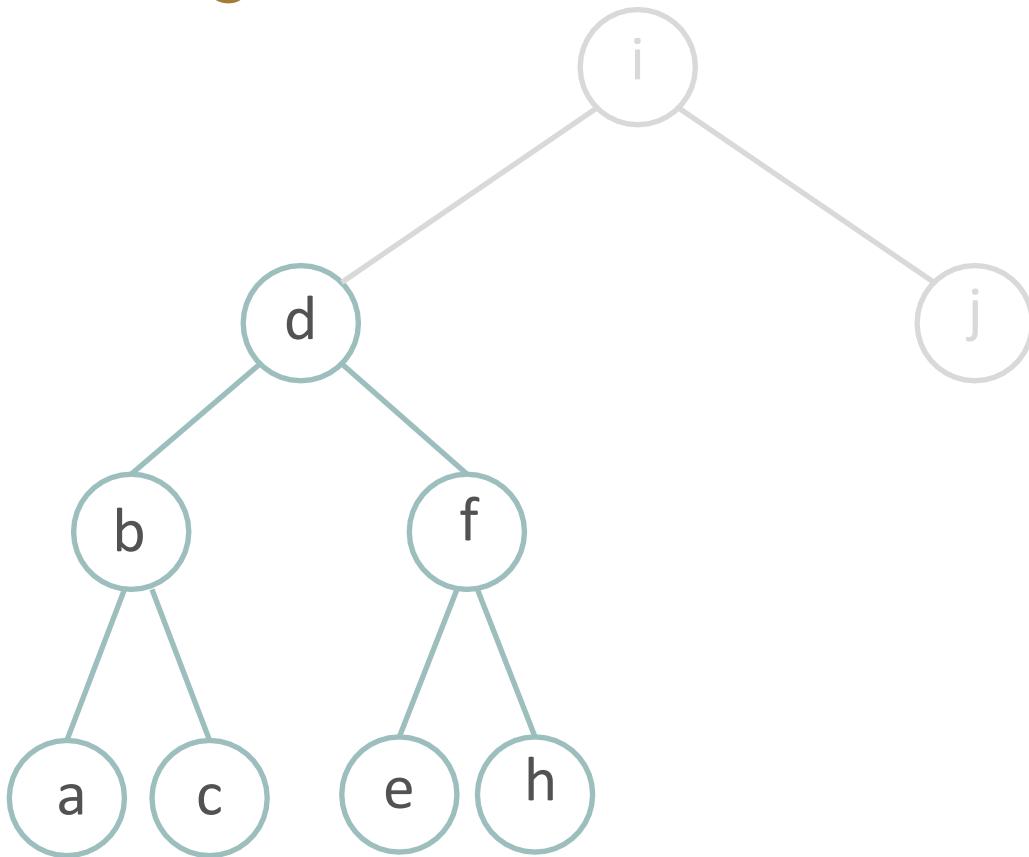


We want to insert **g**

is **g** less than or  
greater than **i**?

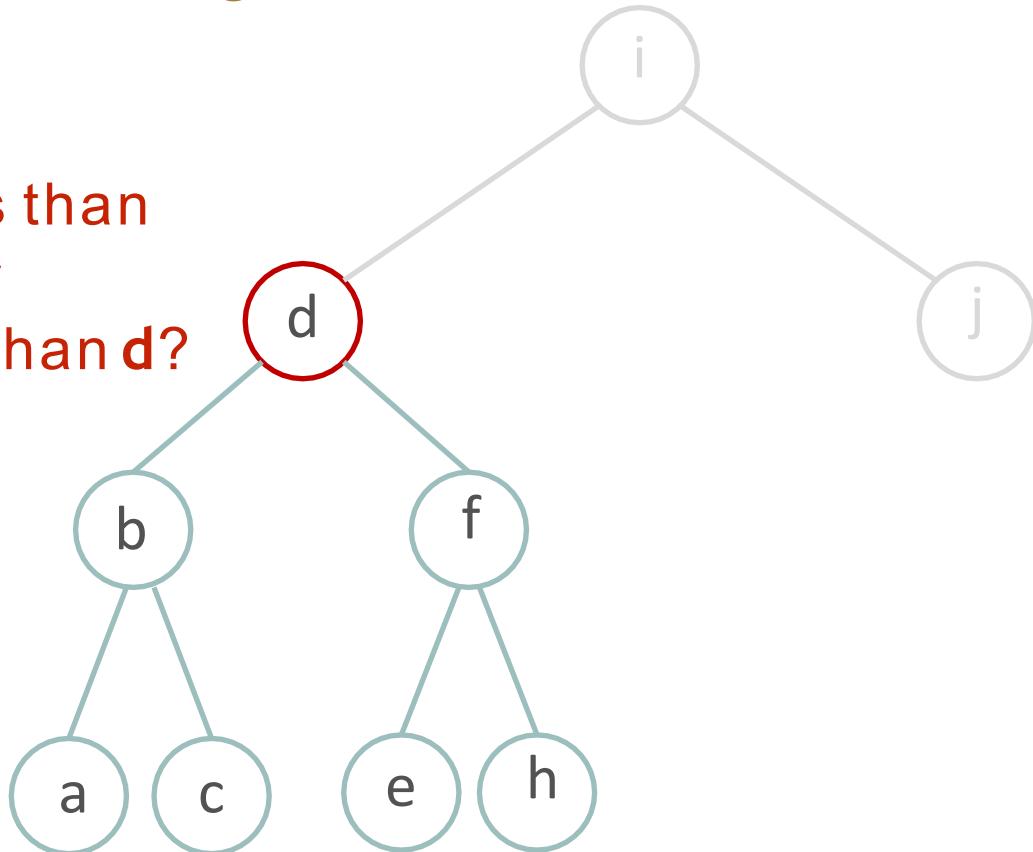


We want to insert **g**

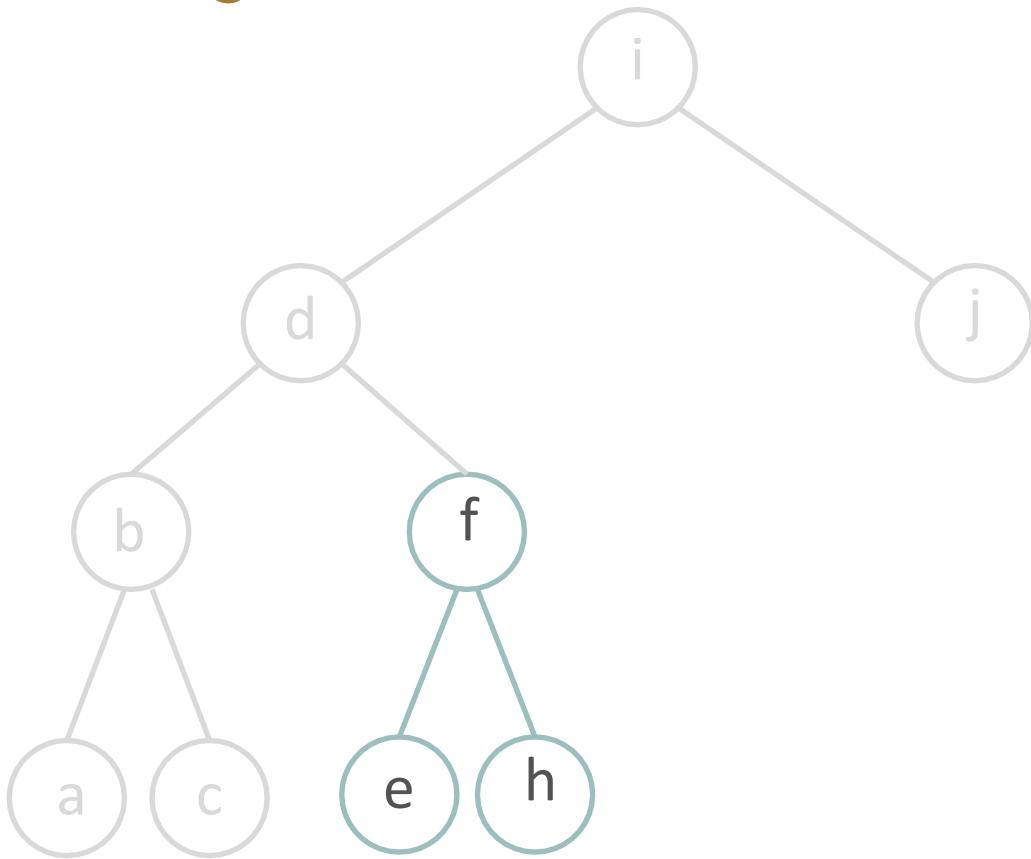


We want to insert **g**

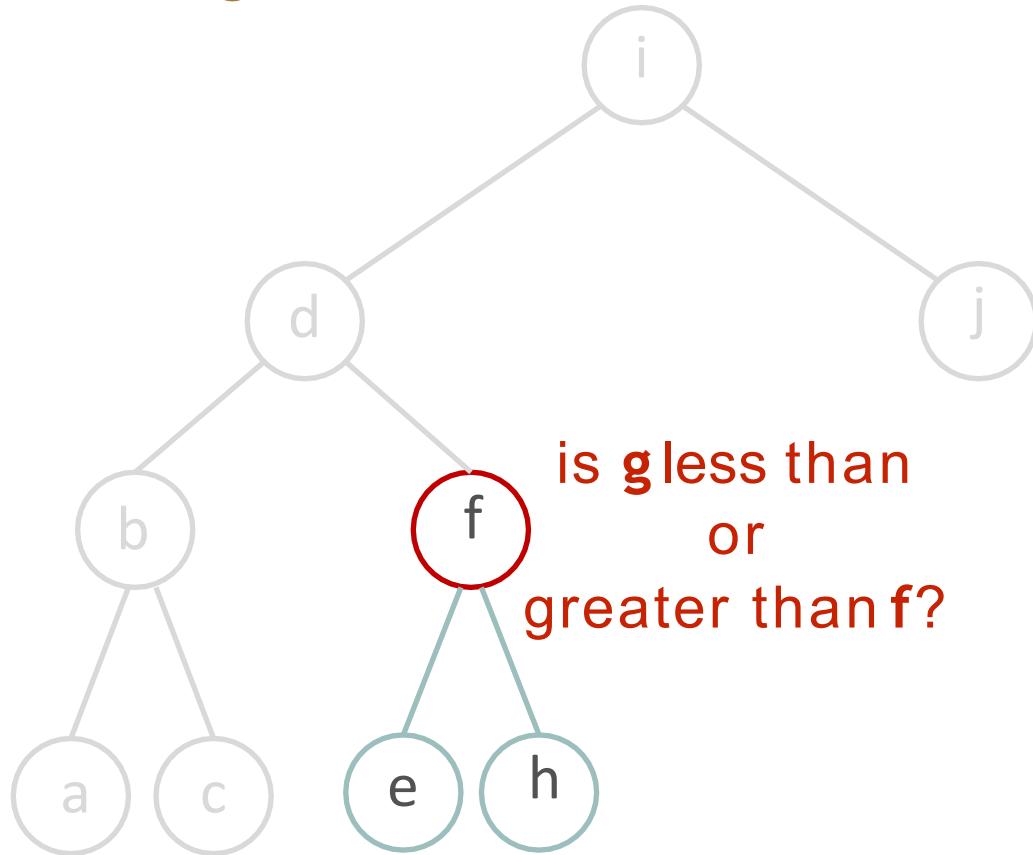
is **g** less than  
or  
greater than **d**?



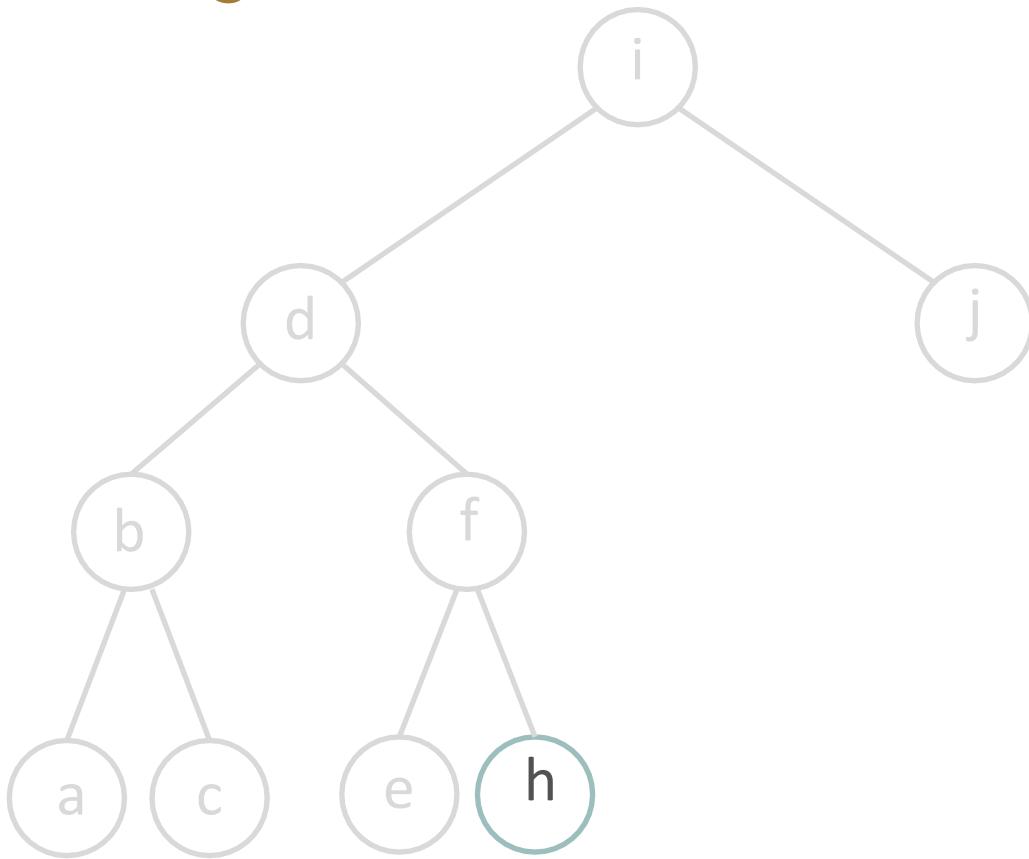
We want to insert **g**



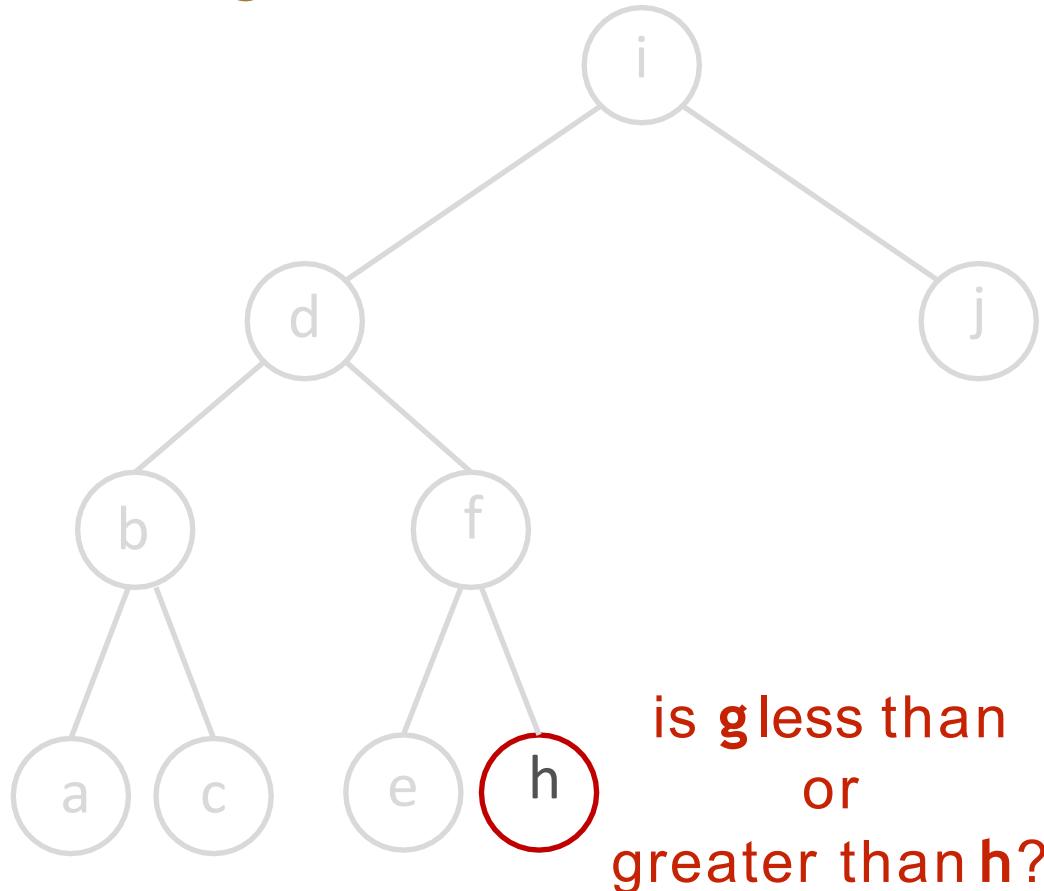
We want to insert **g**



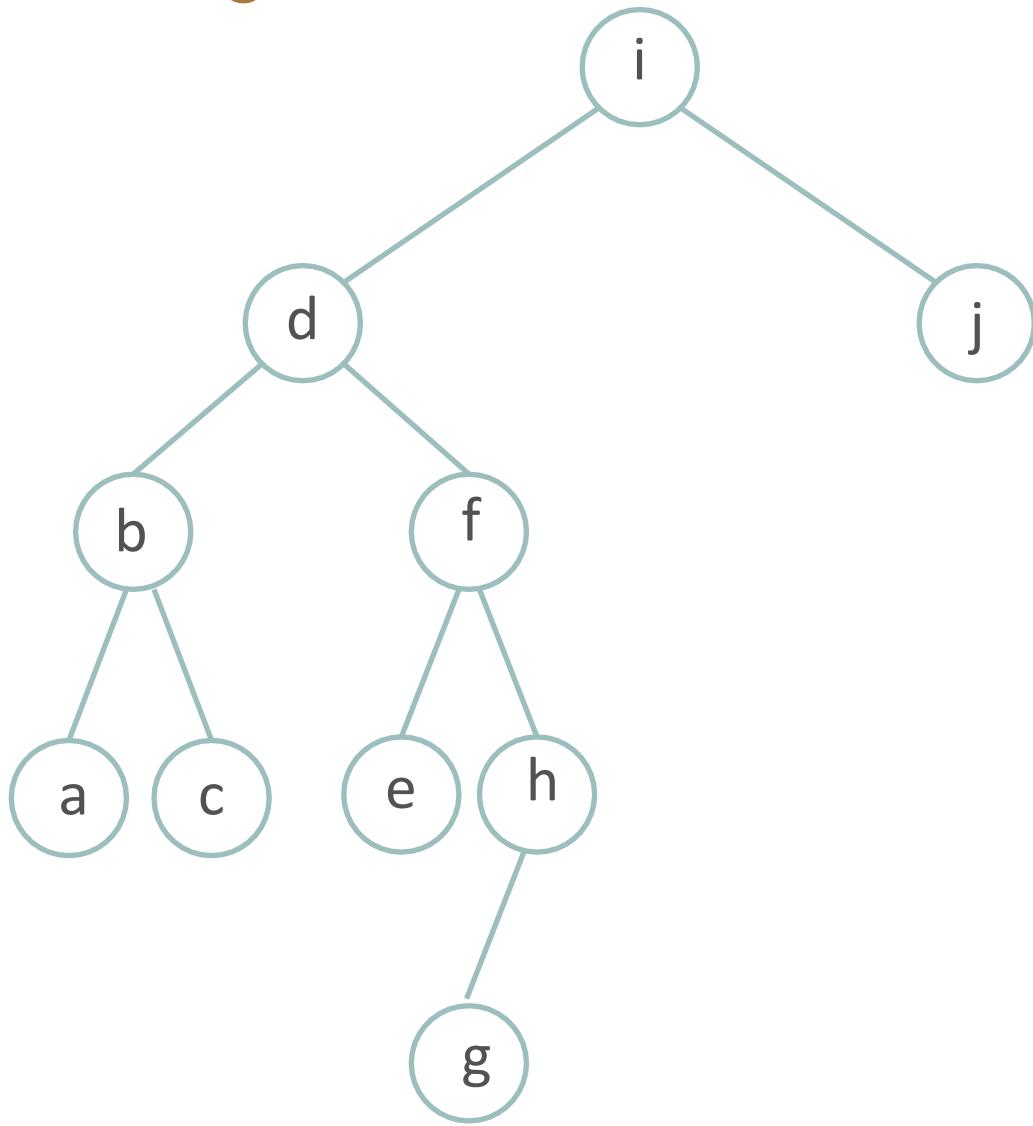
We want to insert **g**



We want to insert **g**

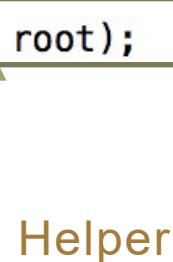


We want to insert **g**



# How does this look like?

```
// Driver method for insertion (start recursion at root)
public void insert(String s)
{
    if(root == null)
        root = new BinaryNode(s);
    else
        insertRecursive(s, root);
}
```



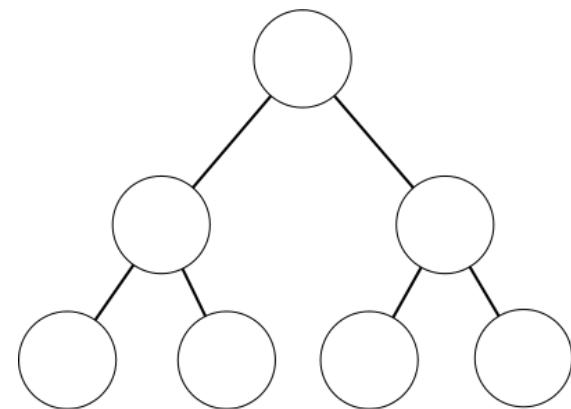
Helper method

# How does this look like?

```
// Driver method for insertion (start recursion at root)
public void insert(String s)
{
    if(root == null)
        root = new BinaryNode(s);
    else
        insertRecursive(s, root);
}
```

---

```
// Recursive method for insertion
private void insertRecursive(String s, BinaryNode n)
{
    // Does the new item go to the left or right?
    if(s.compareTo(n.data) < 0)
    {
        // If we found an empty spot (null), put the item there
        if(n.left == null)
            n.left = new BinaryNode(s);
        else
            insertRecursive(s, n.left);
    }
    else
    {
        // If we found an empty spot (null), put the item there
        if(n.right == null)
            n.right = new BinaryNode(s);
        else
            insertRecursive(s, n.right);
    }
}
```



# Performance

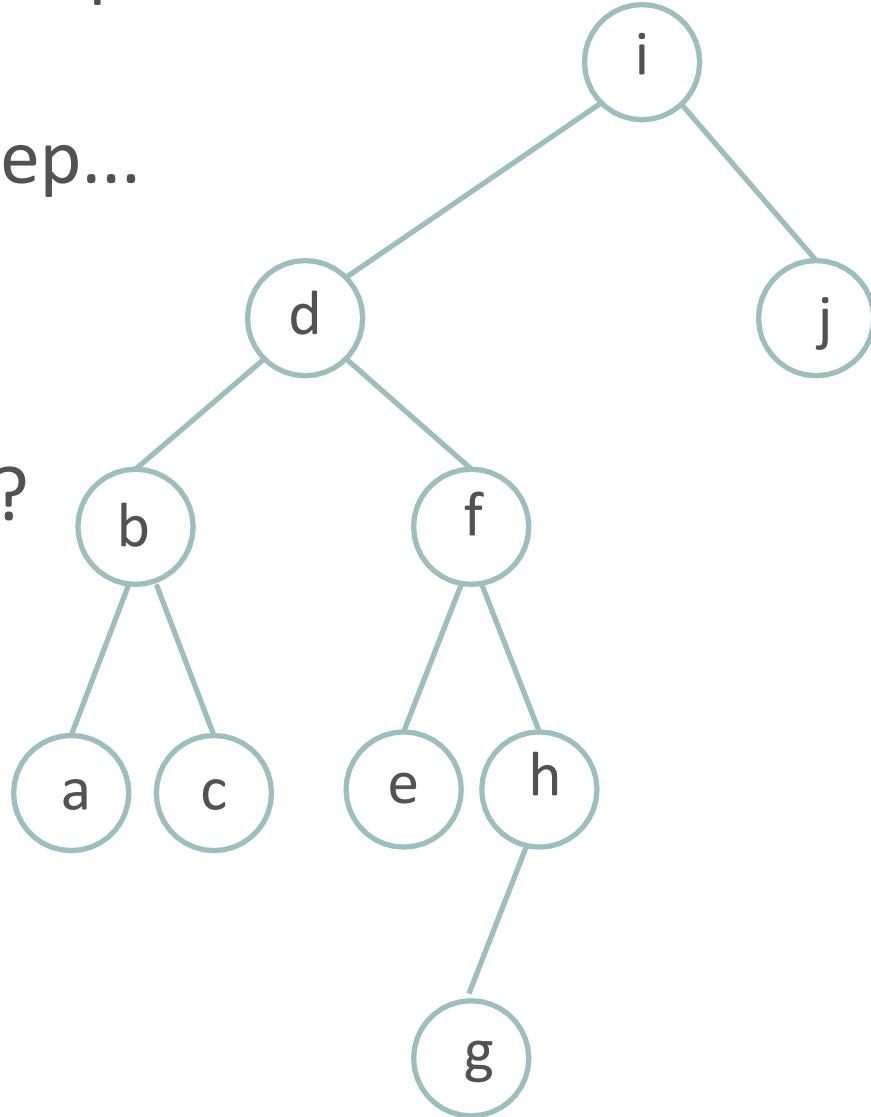
- Big-O complexity of BST searching
  - We disregard one subtree at each step
  - *Roughly half of the remaining nodes!*
- $O(\log N)$

- Big-O complexity of BST searching
  - We disregard one subtree at each step
  - *Roughly half of the remaining nodes!*
- $O(\log N)$
- What is the big-O complexity of BST insertion?

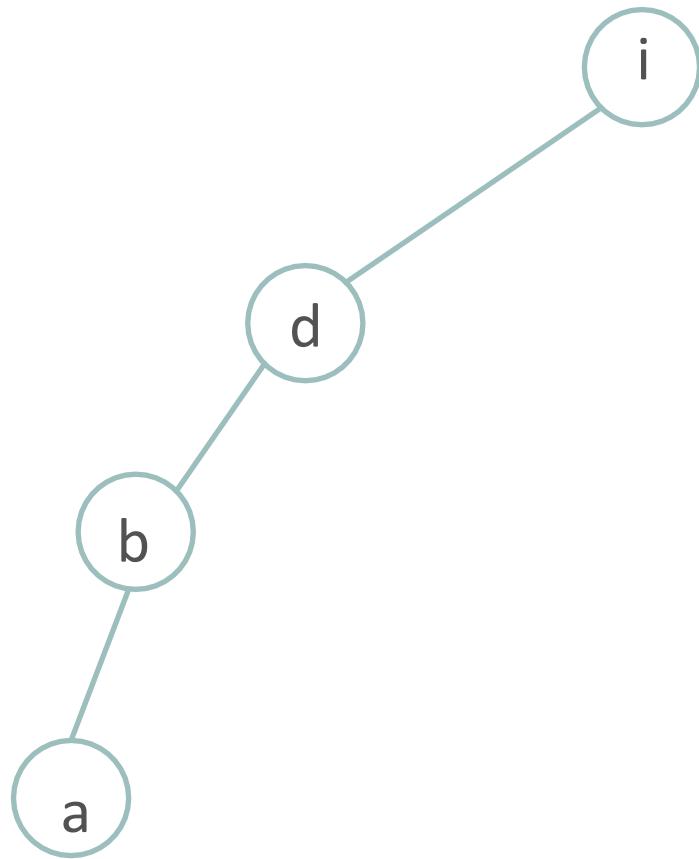
- Big-O complexity of BST searching
  - We disregard one subtree at each step
  - *Roughly half of the remaining nodes!*
- $O(\log N)$
- What is the big-O complexity of BST insertion?
- Searching and insertion are both  $O(\log N)$

- $O(\log N)$  performance requires eliminating half of the possibilities on each step...

- Are all left and right subtrees the same size?

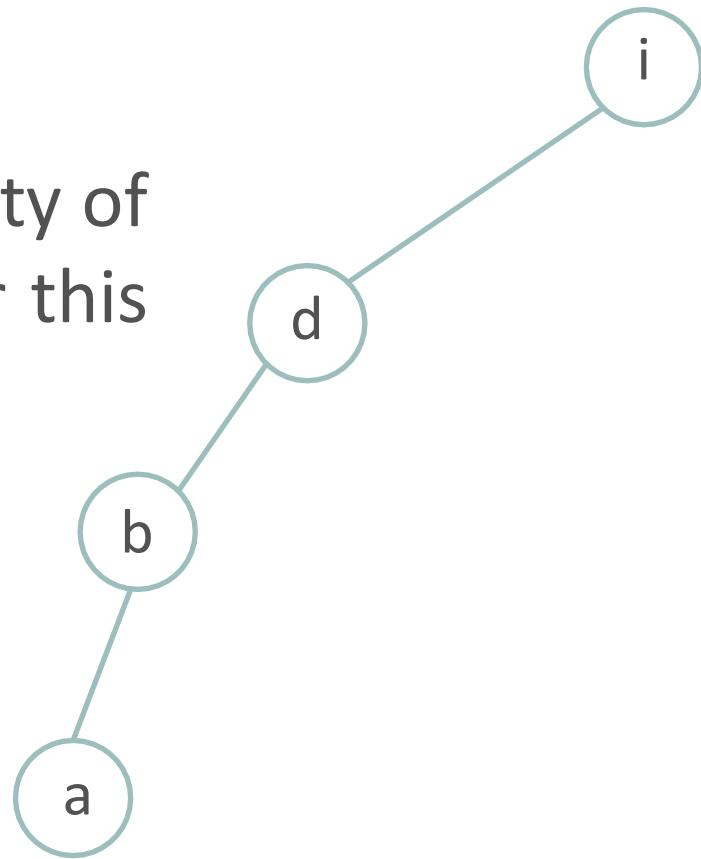


- Is this a valid BST?



- Is this a valid BST?

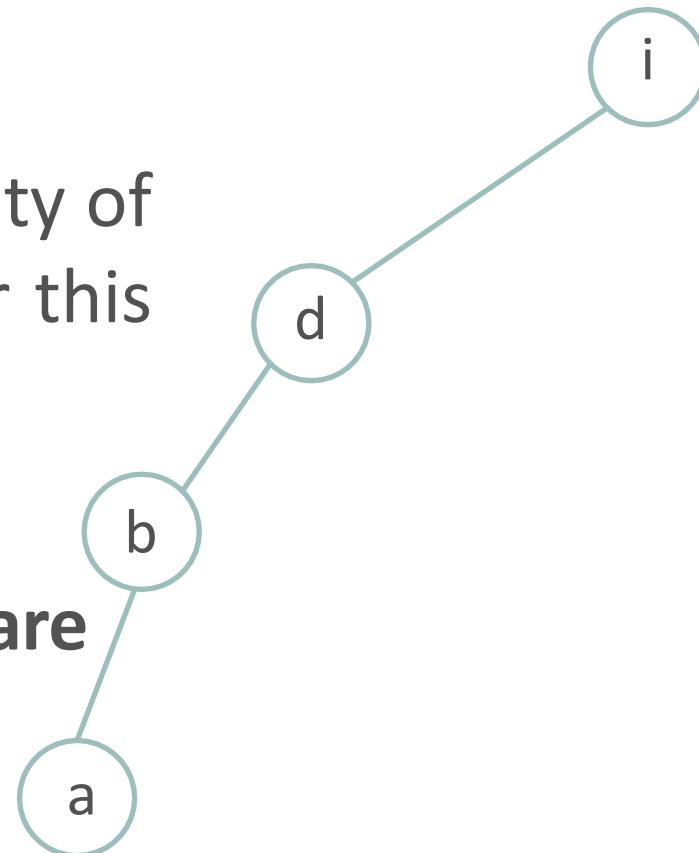
- What is the big-O complexity of searching and insertion for this tree?



- Is this a valid BST?

- What is the big-O complexity of searching and insertion for this tree?

- **The order in which nodes are inserted determines the structure of the tree**



- These nodes were inserted in descending order

# Insertion & searching

- **Worst case:  $O(N)$** 
  - Inserted in ascending or descending order
- **Best case:  $O(\log N)$**
- How does this compare to a sorted array?

# BST vs array

- Arrays can have  $O(\log N)$  searching performance as well
  - How?
- What is the cost of inserting into an array?
  - $O(N)$

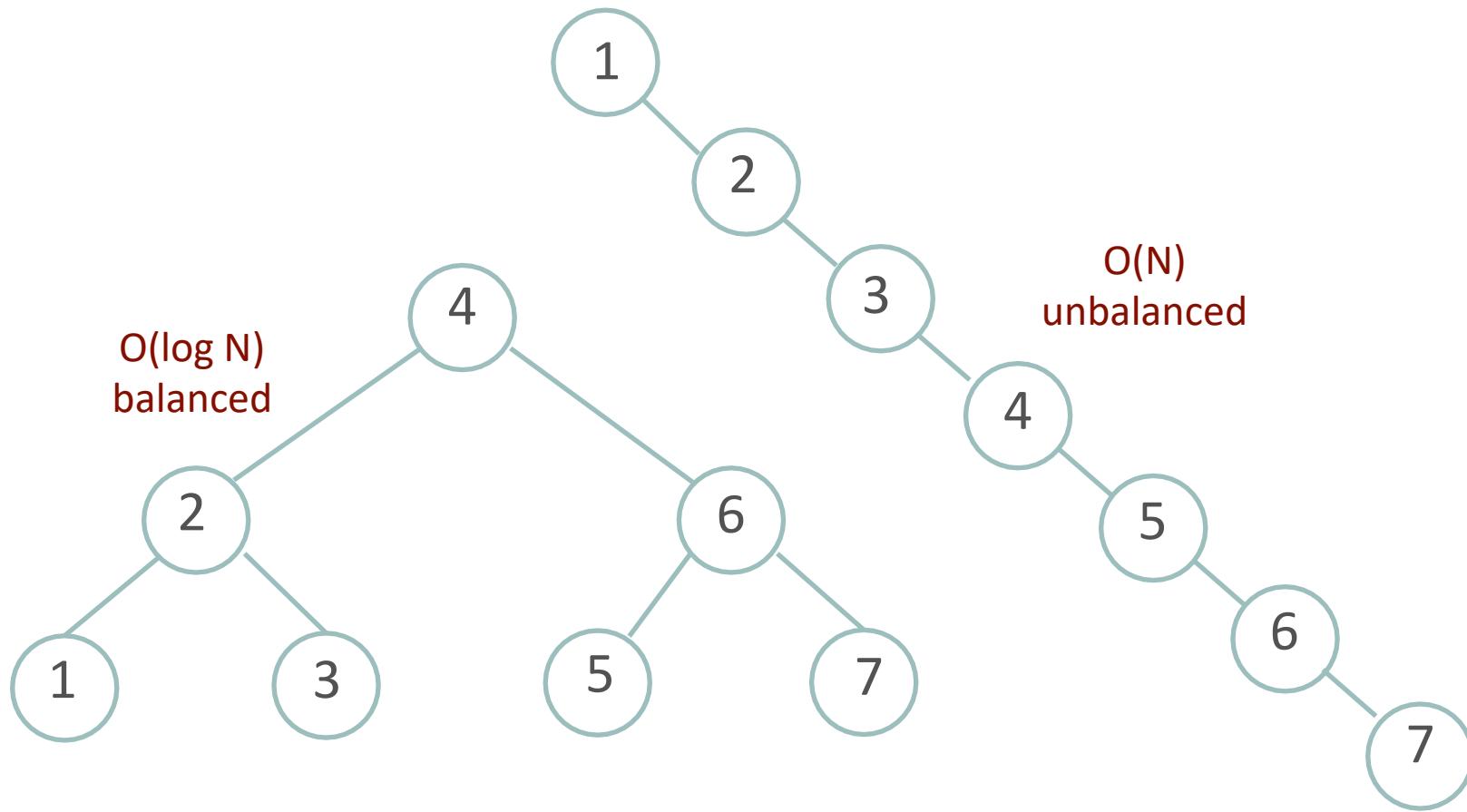
# BST vs array

- Arrays can have  $O(\log N)$  searching performance as well
  - How?
- What is the cost of inserting into an array?
  - $O(N)$

BST wins for insertion!

# balanced vs unbalanced

All operations depend on how well-balanced the tree is



# Deletion

- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

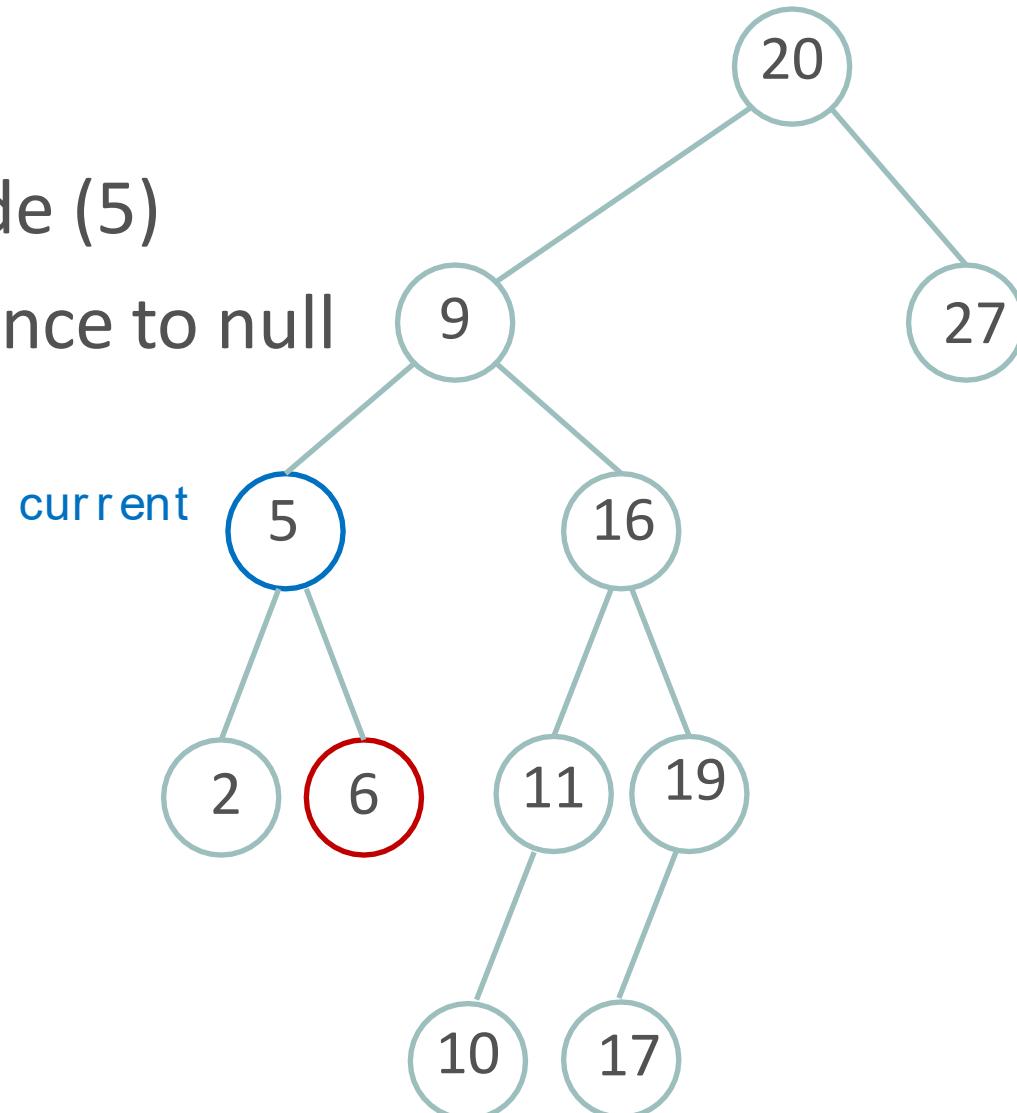
- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list
- There are three different cases:
  - 1) Deleting a leaf node
  - 2) Deleting a node with one child subtree
  - 3) Deleting a node with two children subtrees

- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list
- There are three different cases:
  - 1) Deleting a leaf node
  - 2) Deleting a node with one child subtree
  - 3) Deleting a node with two children subtrees
- First step of deletion is to find the node to delete
  - Just a regular BST search
  - BUT, stop at the *parent* of the node to be deleted

# case 1: Deleting a leaf node

## DELETE NODE 6

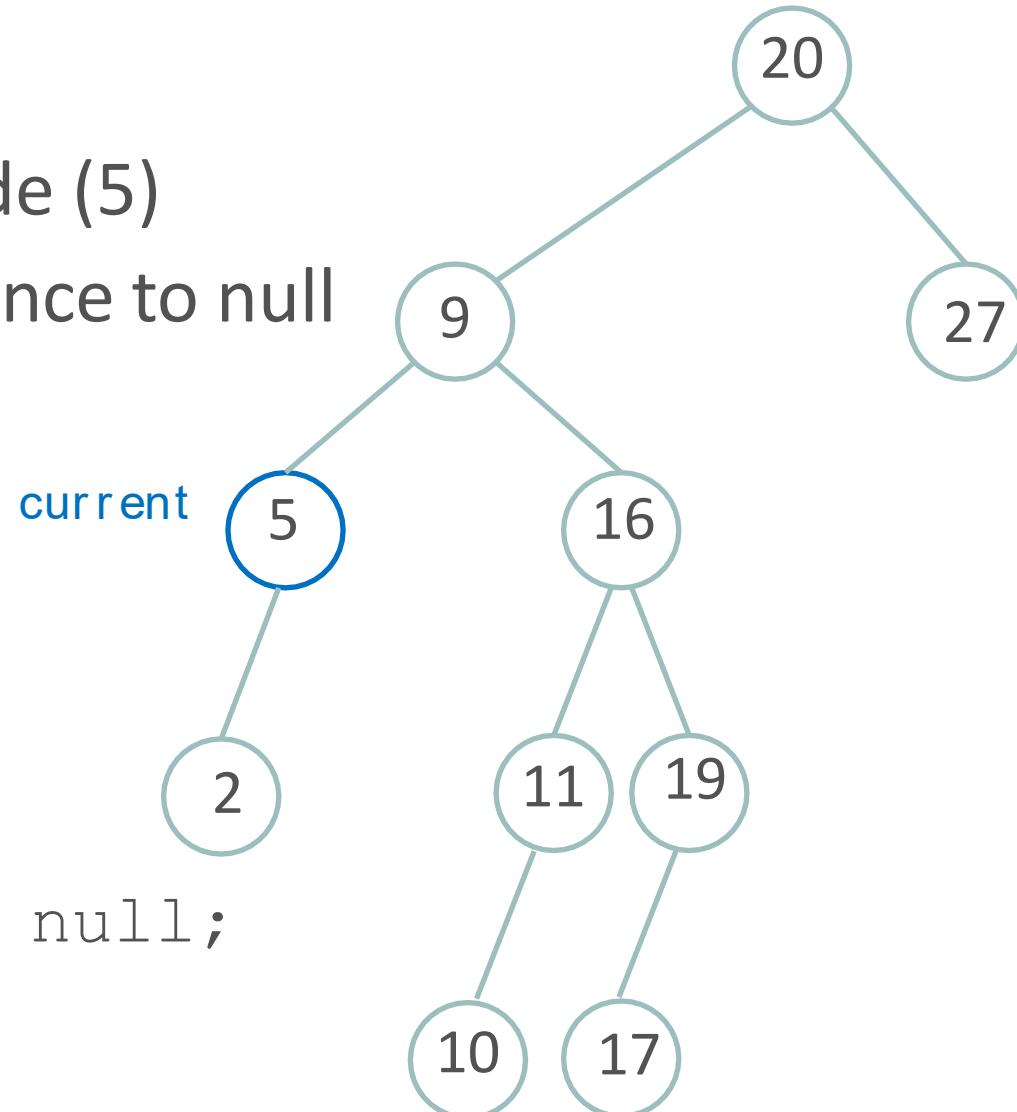
- Stop at parent node (5)
- Set parent's reference to null



# case 1: Deleting a leaf node

## DELETE NODE 6

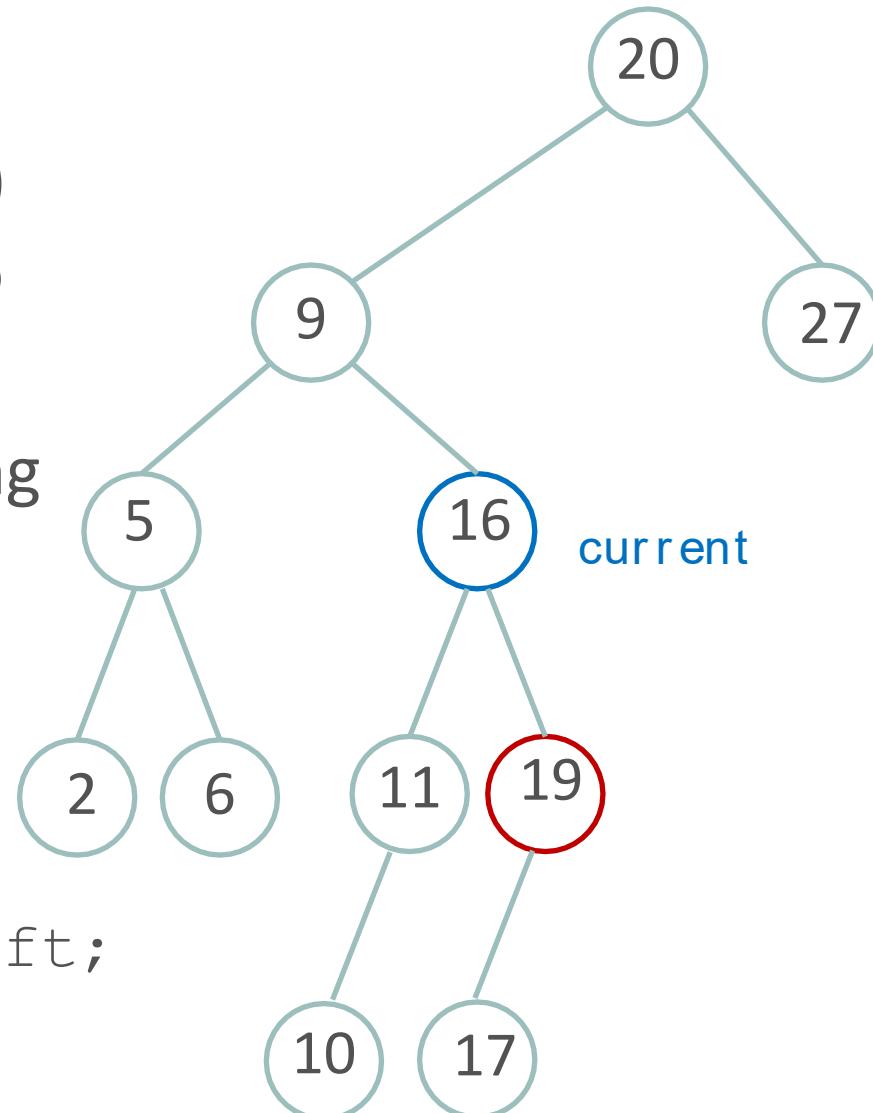
- Stop at parent node (5)
- Set parent's reference to null



# case 2: Delete node with 1 child

## DELETE NODE 19

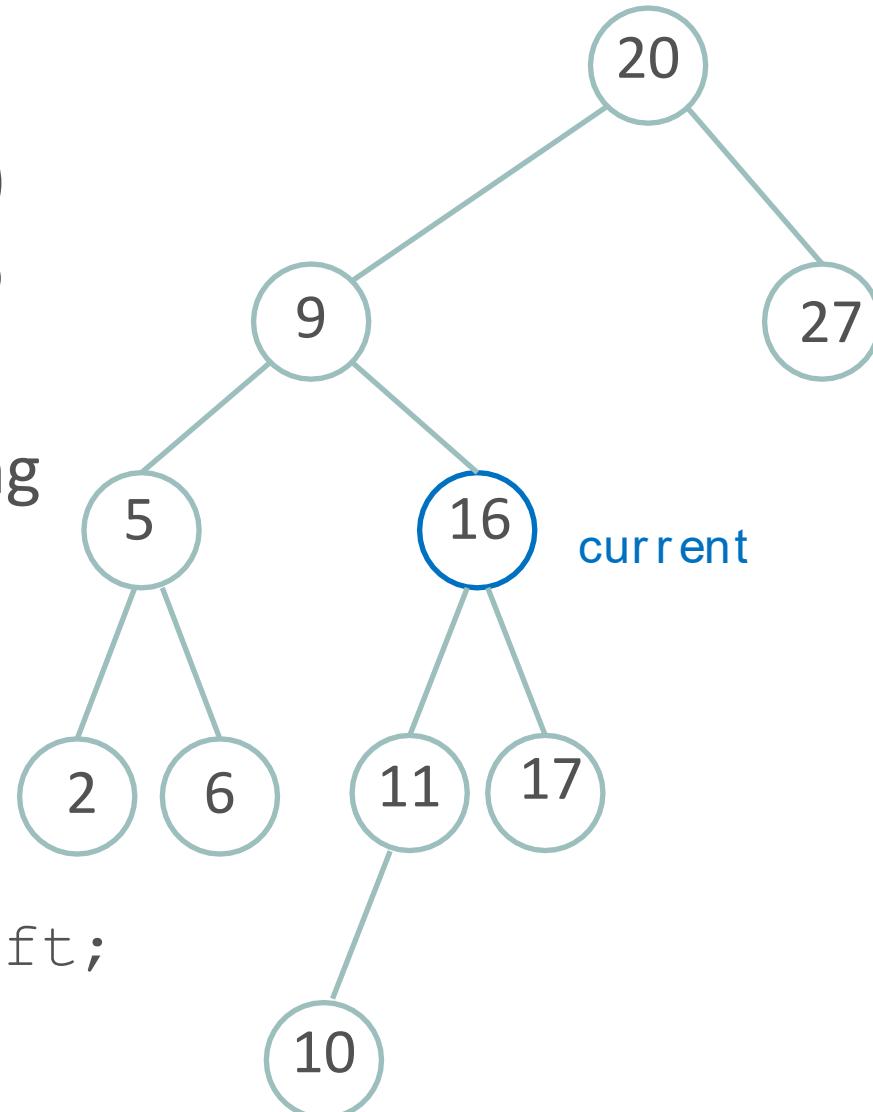
- Stop at parent node (16)
- Set parent's reference to node's child
- Multiple cases depending on which side the child and grandchild are on!



# case 2: Delete node with 1 child

## DELETE NODE 19

- Stop at parent node (16)
- Set parent's reference to node's child
- Multiple cases depending on which side the child and grandchild are on!

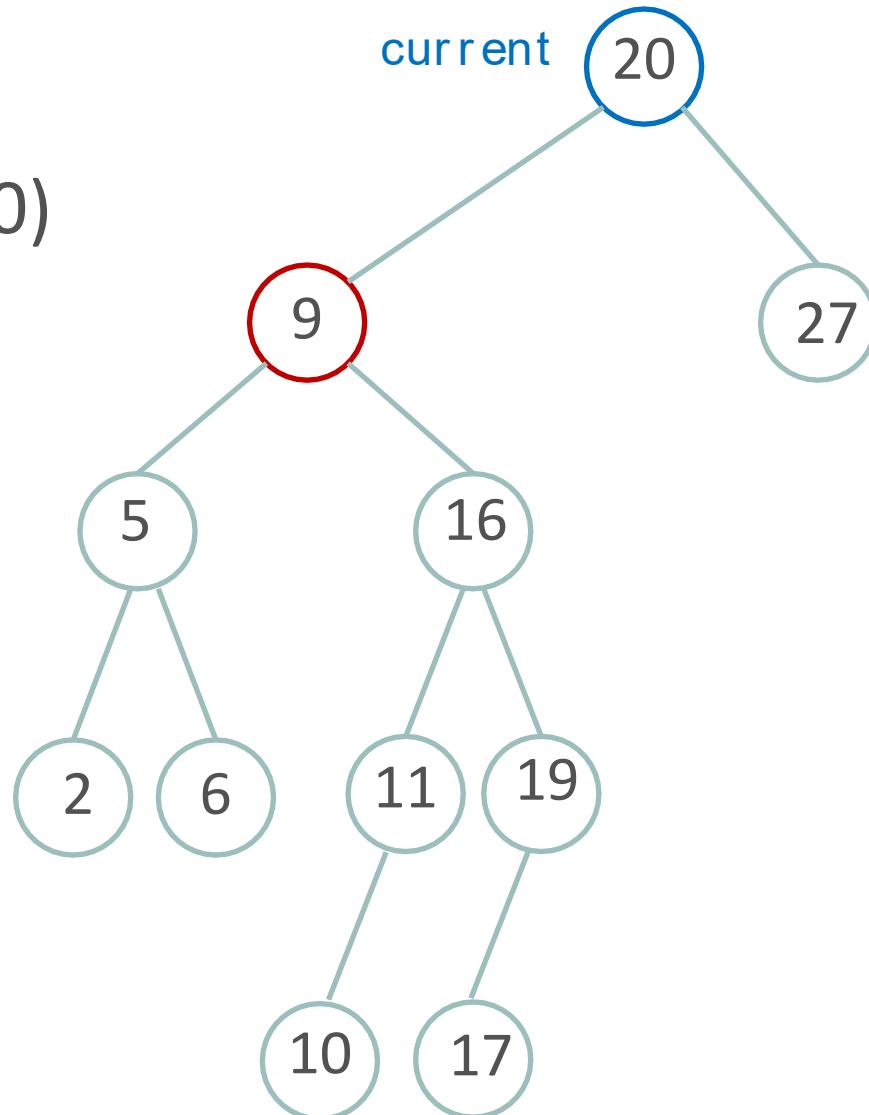


```
current.right =  
    current.right.left;
```

# case 3: Delete node with 2 children

## DELETE NODE 9

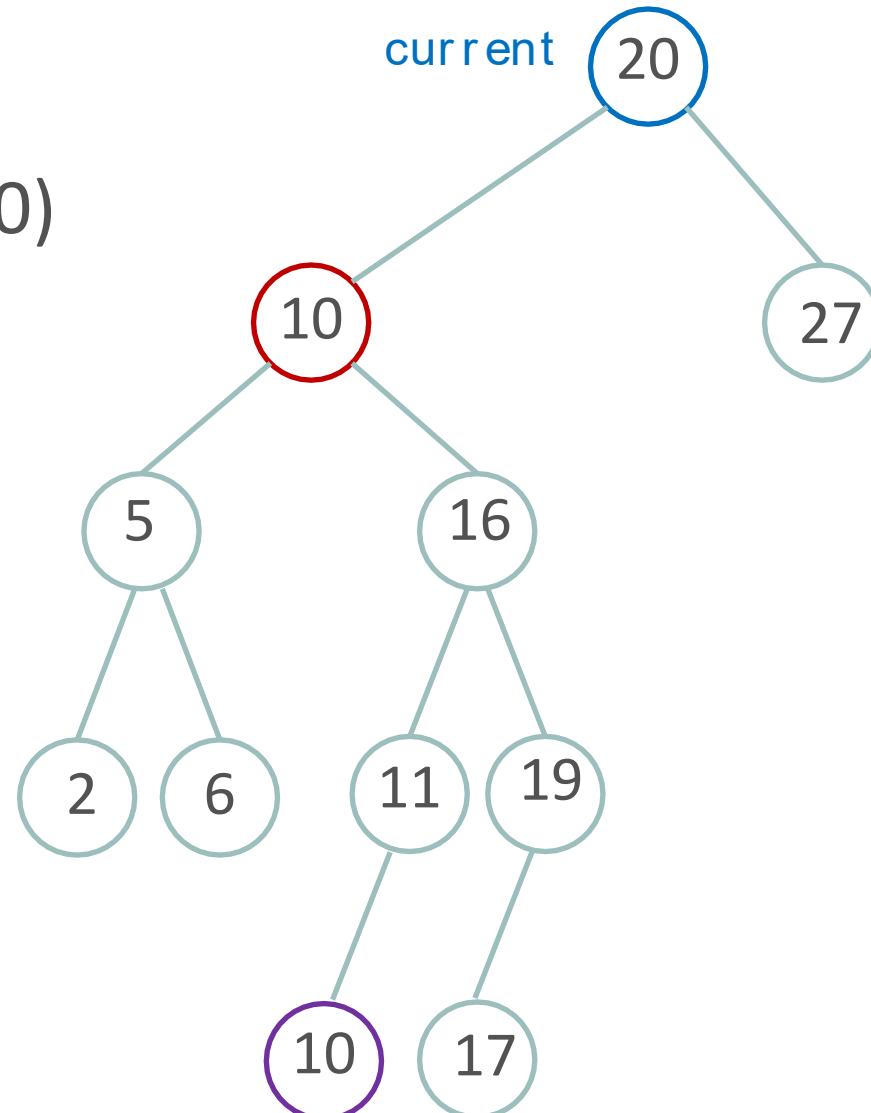
- Stop at parent node (20)
- Replace the node with the smallest item in its right subtree(10)



# case 3: Delete node with 2 children

## DELETE NODE 9

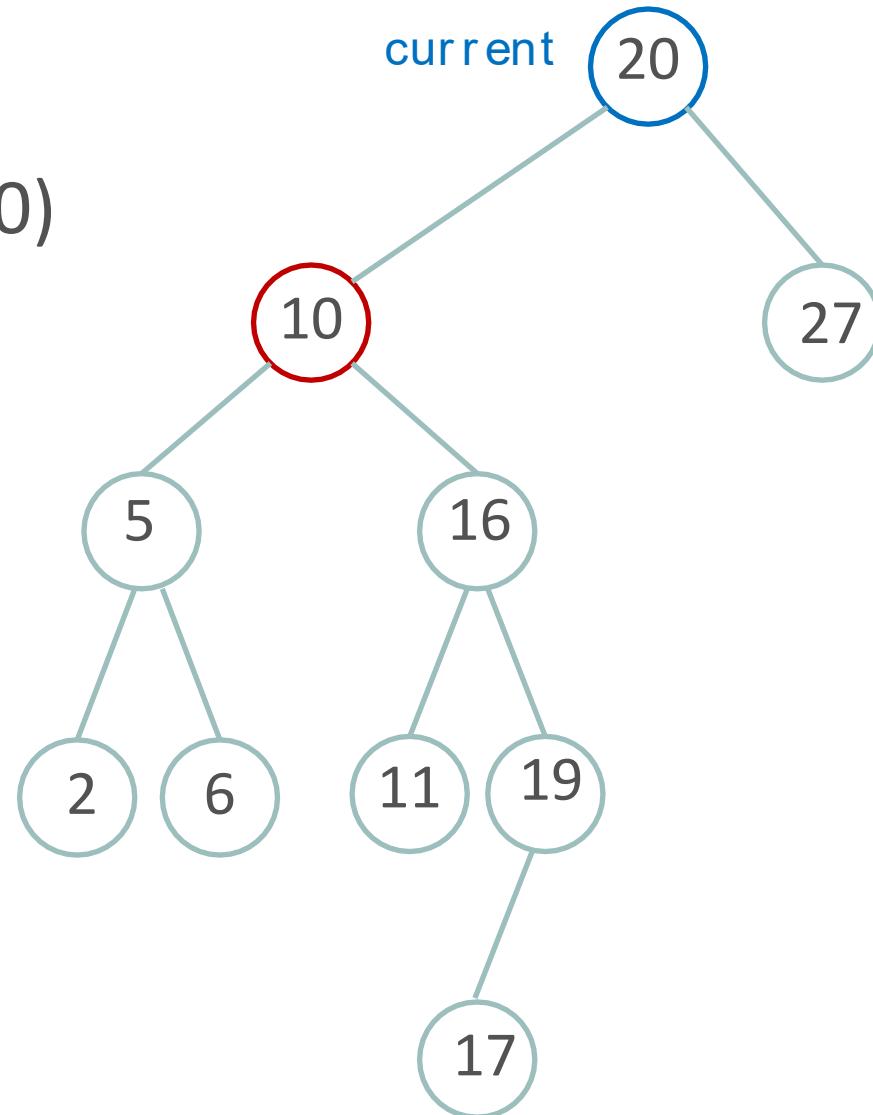
- Stop at parent node (20)
- Replace the node with the smallest item in its right subtree(10)
- Perform a deletion on successor (10)



# case 3: Delete node with 2 children

## DELETE NODE 9

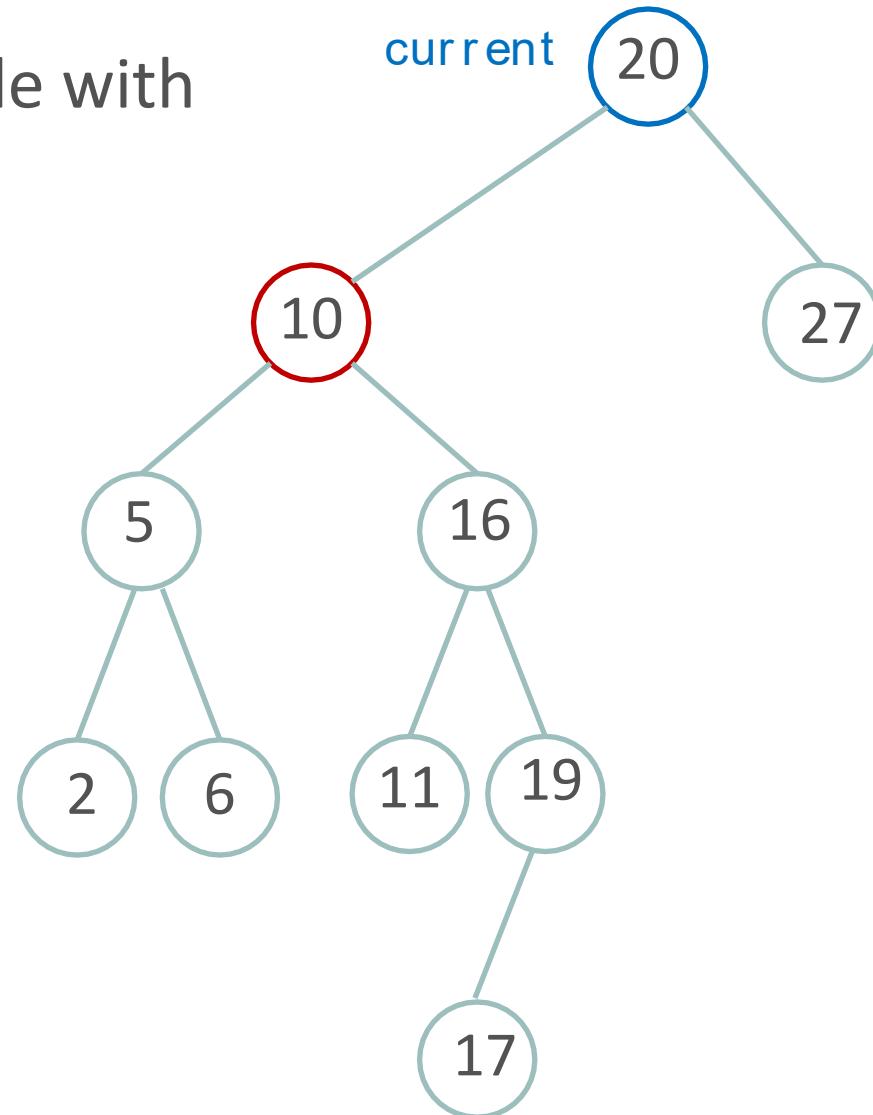
- Stop at parent node (20)
- Replace the node with the smallest item in its right subtree(10)
- Perform a deletion on successor (10)



# case 3: Delete node with 2 children

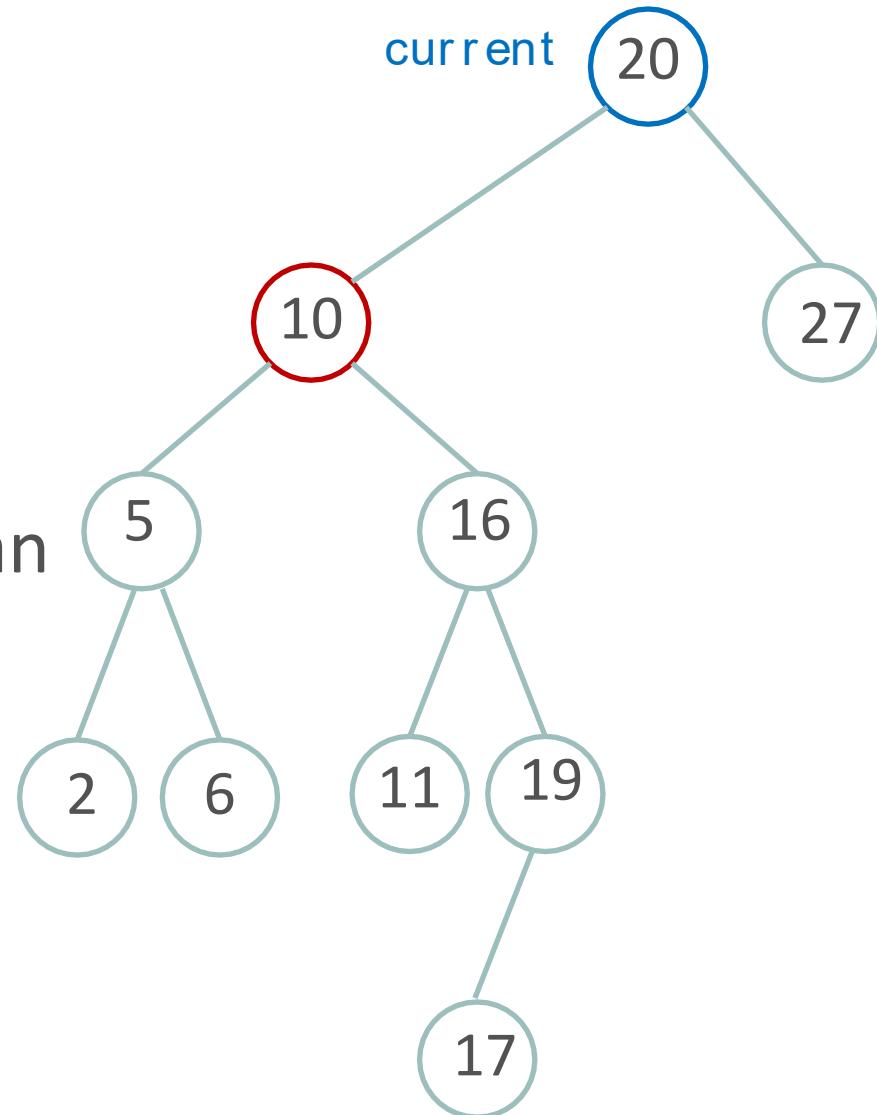
Replacing the deleted node with  
the smallest child in right  
subtree guarantees  
the BST structure...

Why?



# case 3: Delete node with 2 children

The smallest item in the right subtree is greater than any item in the left subtree, *and* smaller than any item in the new right subtree.

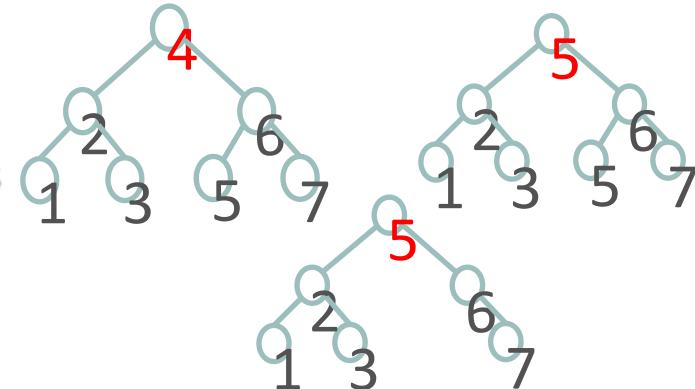
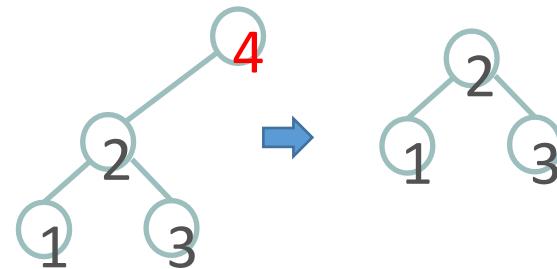
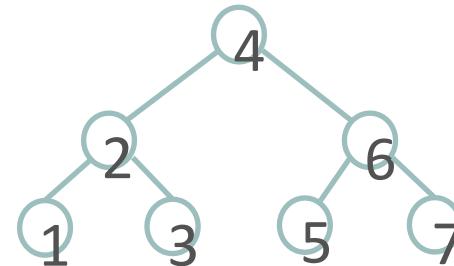


```
1 // Java program to demonstrate delete operation in binary search tree
2 class BinarySearchTree
3 {
4     /* Class containing left and right child of current node and key value*/
5     class Node
6     {
7         int key;
8         Node left, right;
9
10        public Node(int item)
11        {
12            key = item;
13            left = right = null;
14        }
15    }
16
17    // Root of BST
18    Node root;
19
20    // Constructor
21    BinarySearchTree()
22    {
23        root = null;
24    }
25
26    // This method mainly calls deleteRec()
27    void deleteKey(int key)
28    {
29        root = deleteRec(root, key);
30    }
```

```

31
32  /* A recursive function to delete a node with given key */
33  Node deleteRec(Node root, int key)
34  {
35      /* Base Case: If the tree is empty */
36      if (root == null) return root;
37
38      /* Otherwise, recur down the tree */
39      if (key < root.key)
40          root.left = deleteRec(root.left, key);
41      else if (key > root.key)
42          root.right = deleteRec(root.right, key);
43
44      // if key is same as root's key, then This is the node
45      // to be deleted
46      else
47      {
48          // node with only one child or no child
49          if (root.left == null)
50              return root.right;
51          else if (root.right == null)
52              return root.left;
53
54          // node with two children: Get the inorder successor (smallest
55          // in the right subtree)
56          root.key = minValue(root.right);
57
58          // Delete the inorder successor
59          root.right = deleteRec(root.right, root.key);
60      }
61
62      return root;
63  }

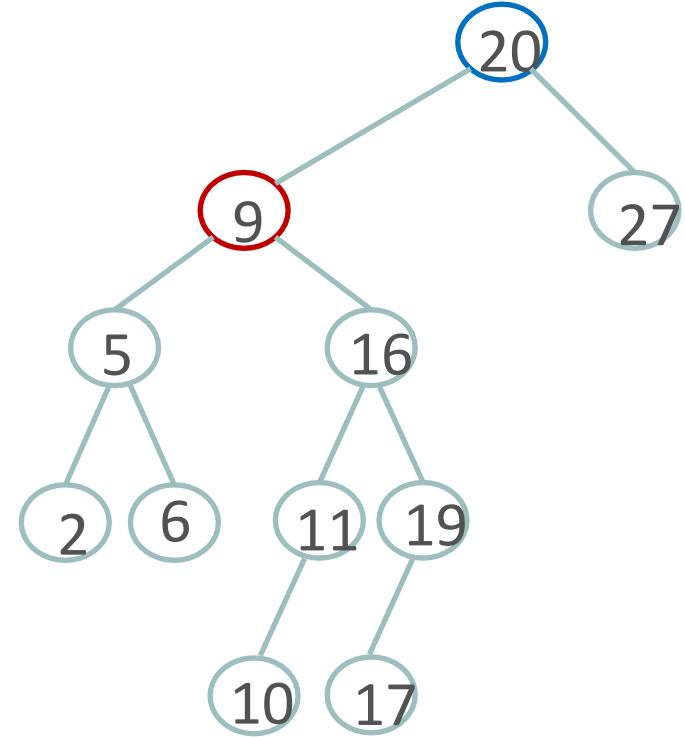
```



```

65     int minValue(Node root)
66     {
67         int minv = root.key;
68         while (root.left != null)
69         {
70             minv = root.left.key;
71             root = root.left;
72         }
73         return minv;
74     }
75
76     // This method mainly calls insertRec()
77     void insert(int key)
78     {
79         root = insertRec(root, key);
80     }
81
82     /* A recursive function to insert a new key in BST */
83     Node insertRec(Node root, int key)
84     {
85
86         /* If the tree is empty, return a new node */
87         if (root == null)
88         {
89             root = new Node(key);
90             return root;
91         }
92
93         /* Otherwise, recur down the tree */
94         if (key < root.key)
95             root.left = insertRec(root.left, key);
96         else if (key > root.key)
97             root.right = insertRec(root.right, key);
98
99         /* return the (unchanged) node pointer */
100        return root;
101    }
102
103    // This method mainly calls InorderRec()
104    void inorder()
105    {
106        inorderRec(root);
107    }

```



```
108
109 // A utility function to do inorder traversal of BST
110 void inorderRec(Node root)
111 {
112     if (root != null)
113     {
114         inorderRec(root.left);
115         System.out.print(root.key + " ");
116         inorderRec(root.right);
117     }
118 }
119
120 // Driver Program to test above functions
121 public static void main(String[] args)
122 {
123     BinarySearchTree tree = new BinarySearchTree();
124
125     /* Let us create following BST
126      50
127      /   \
128      30   70
129      / \ / \
130      20 40 60 80 */
131     tree.insert(50);
132     tree.insert(30);
133     tree.insert(20);
134     tree.insert(40);
135     tree.insert(70);
136     tree.insert(60);
137     tree.insert(80);
138
139     System.out.println("Inorder traversal of the given tree");
140     tree.inorder();
141
142     System.out.println("\nDelete 20");
143     tree.deleteKey(20);
144     System.out.println("Inorder traversal of the modified tree");
145     tree.inorder();
146
147     System.out.println("\nDelete 30");
148     tree.deleteKey(30);
149     System.out.println("Inorder traversal of the modified tree");
150     tree.inorder();
151
152     System.out.println("\nDelete 50");
153     tree.deleteKey(50);
154     System.out.println("Inorder traversal of the modified tree");
155     tree.inorder();
156 }
157 }
158 }
```

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:
- Cost of:
  - Case 1 (delete leaf):
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete: **O(log N)**
- Cost of:
  - Case 1 (delete leaf):
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete: **O(log N)**
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null: O(1)**
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:  $O(\log N)$
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null:  $O(1)$**
  - Case 2 (delete node with 1 child):  
**bypass a reference:  $O(1)$**
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:  $O(\log N)$
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null:  $O(1)$**
  - Case 2 (delete node with 1 child):  
**bypass a reference:  $O(1)$**
  - Case 3 (delete node with 2 children):  
**Find the successor:  $O(\log N)$**   
**delete the duplicate successor:  $O(1)$**

# Other useful properties

- How can we find the smallest node?

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree

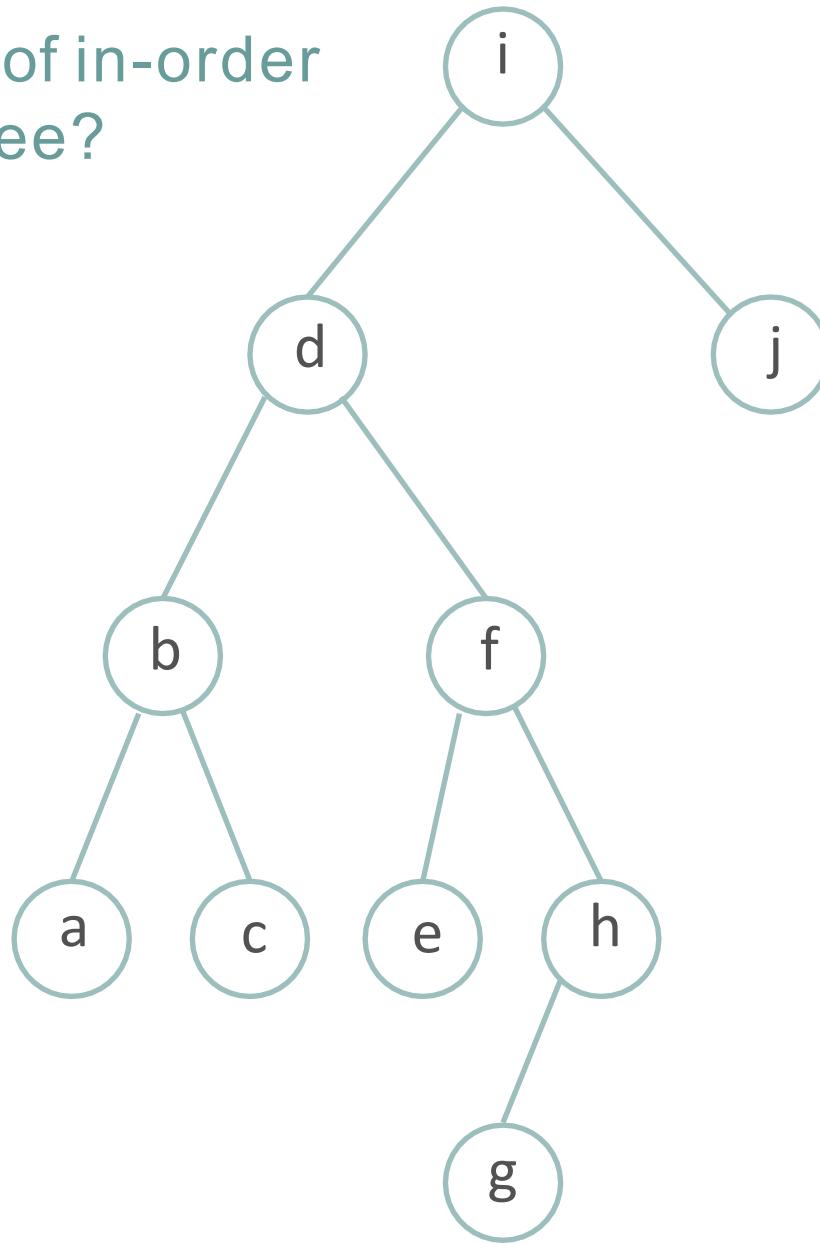
# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree
- How can we print nodes in ascending order?

# Other useful properties

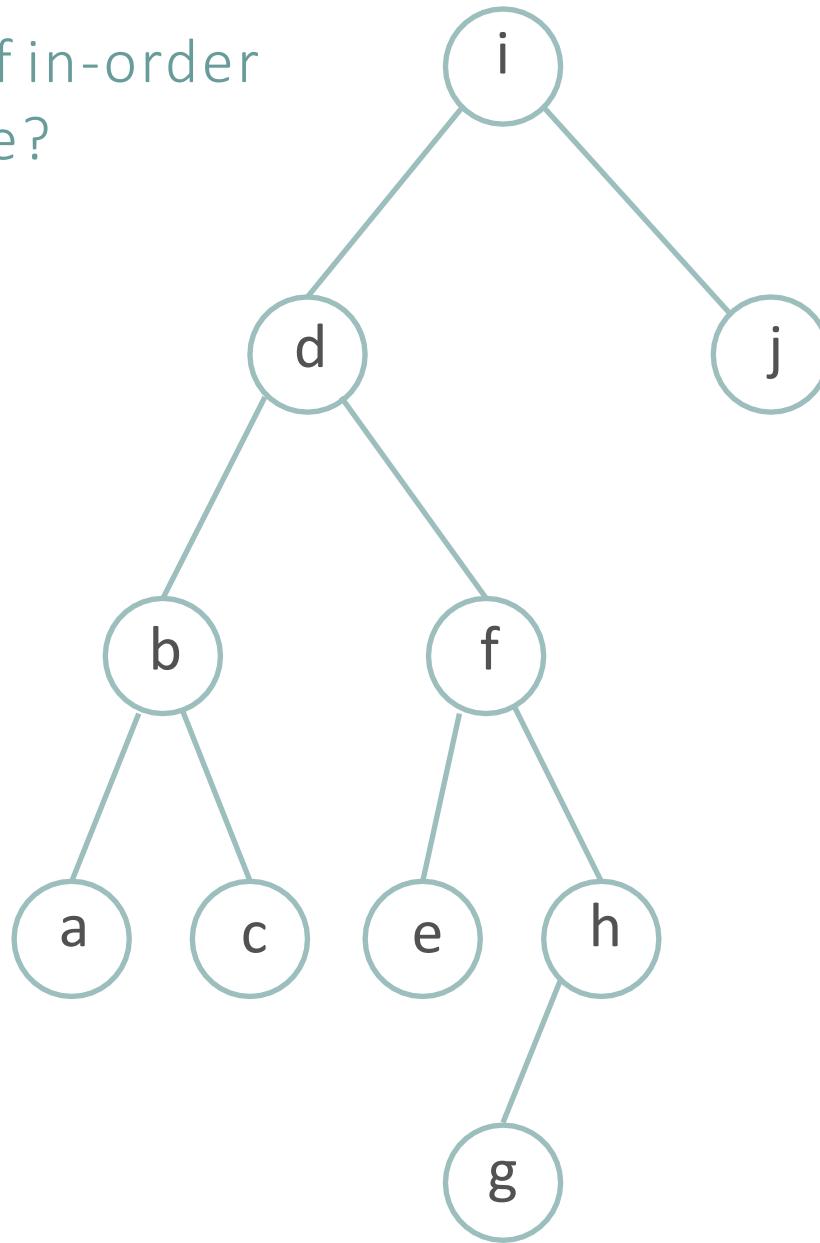
- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree
- How can we print nodes in ascending order?
  - In-order traversal

What is the result of in-order traversal of this tree?



a b c d e f g h i j

What is the result of in-order traversal of this tree?



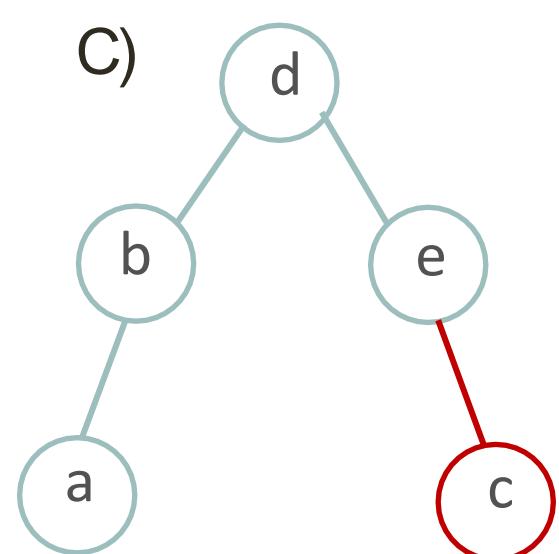
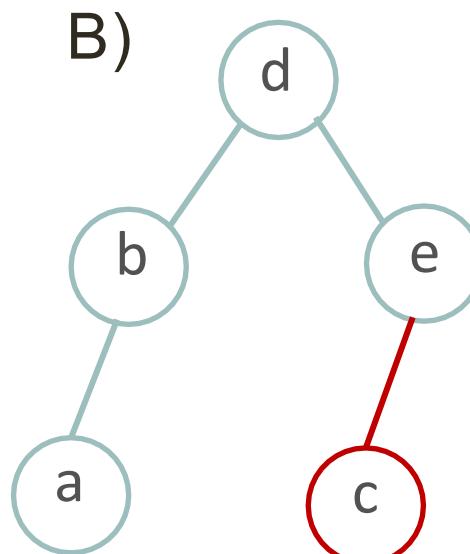
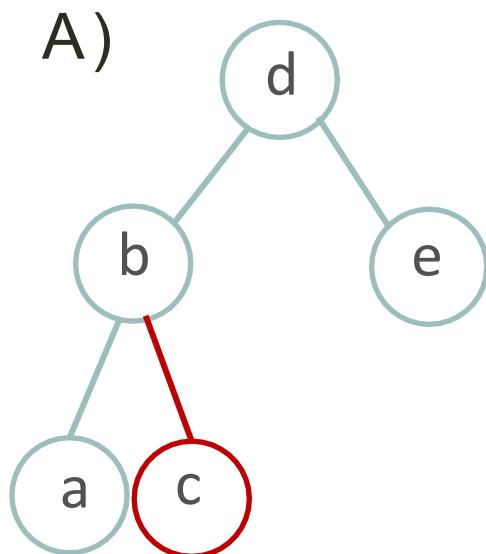
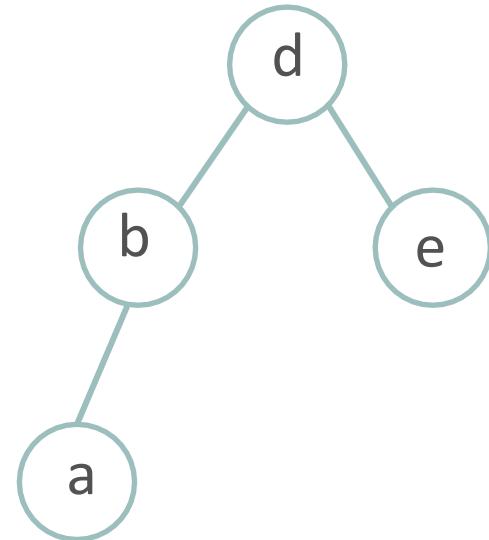
Alphabetical order  
Or  
Lexicographical order



a b c d e f g h i j

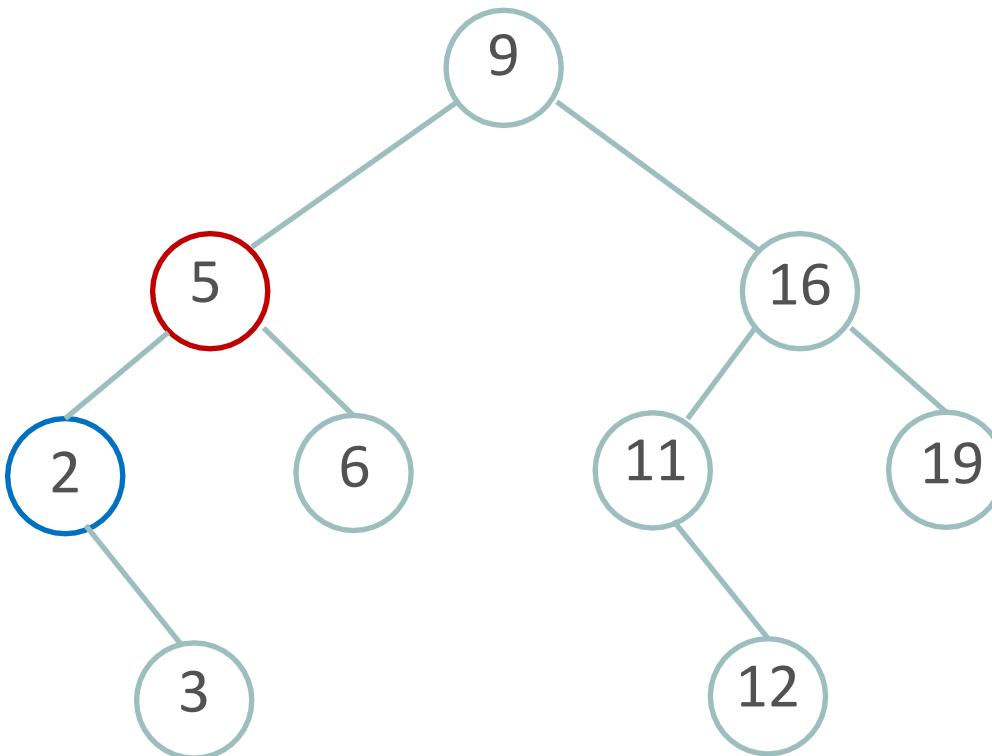
# Review

Which of the following trees is the result of adding c to this bst?



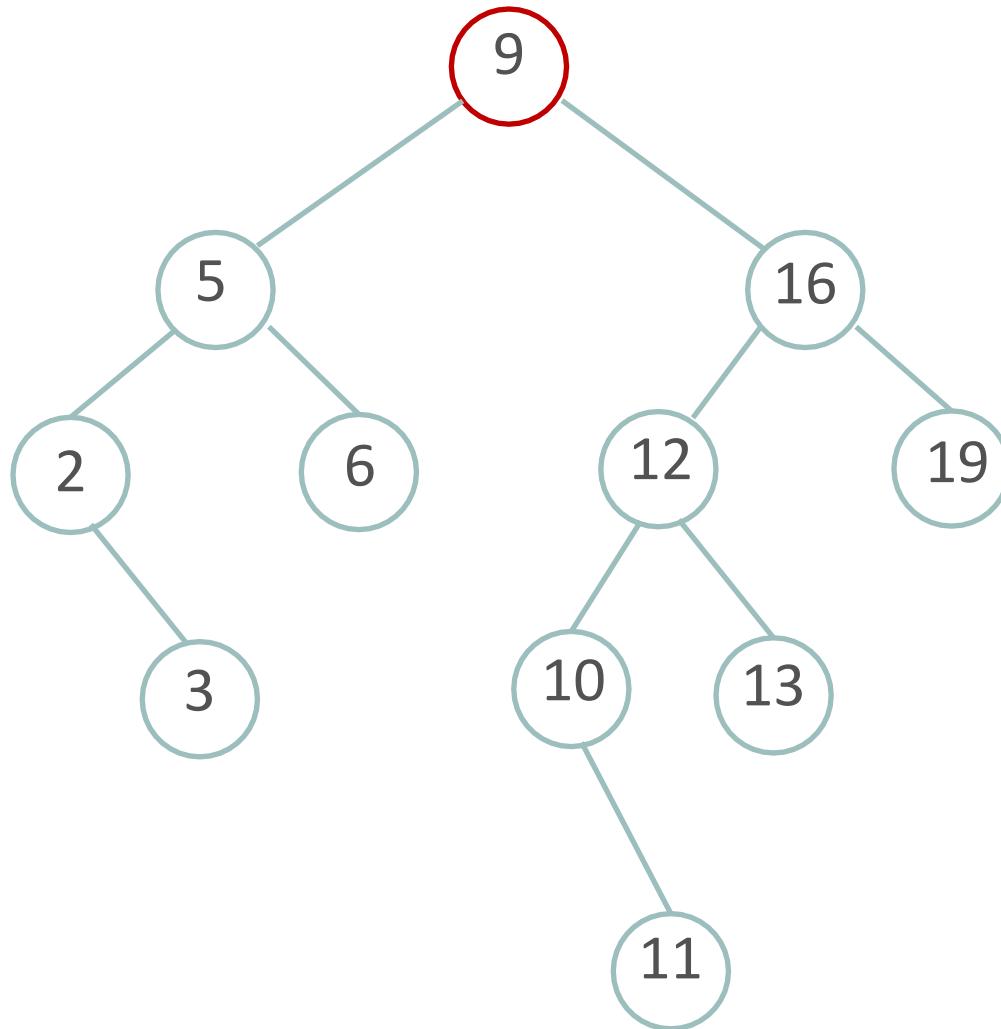
What will 5's left child be after deleting 2?

- A) 3
- B) 6
- C) null



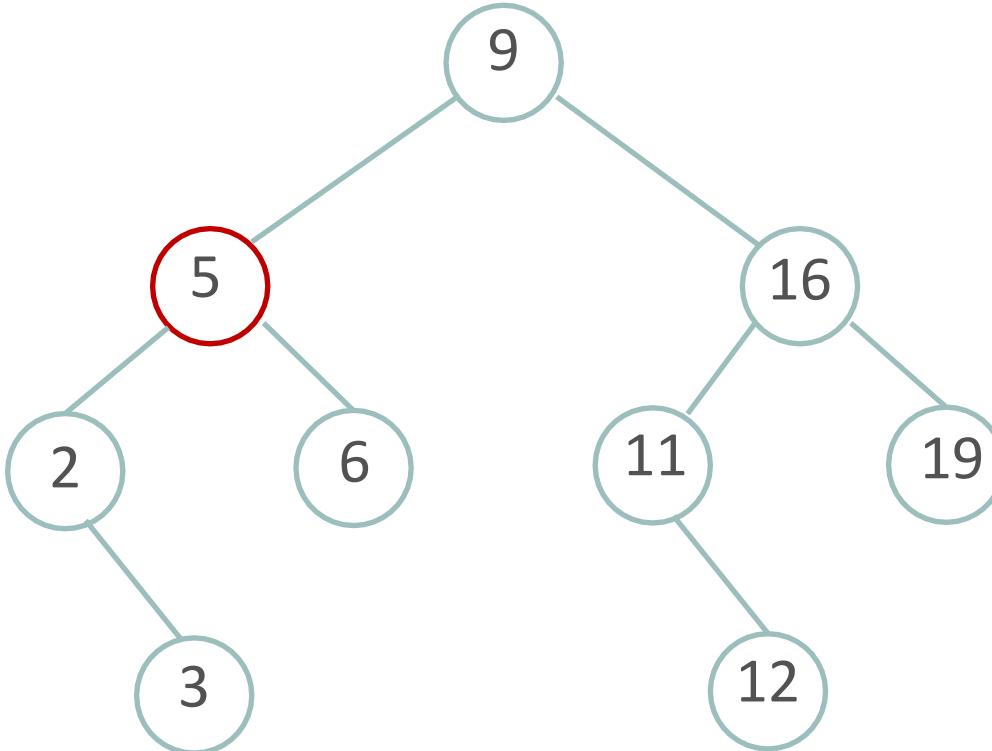
What will 5's left child be after deleting 9?

- A) 6
- B) 10
- C) 13
- D) 19



What will 5's left child be after deleting 2?

- A) 2
- B) 3
- C) 6
- D) 12



Next time...

- Next week
  - Graphs
- Start your assignment early

Hash Tables  
CSC220 | Computer Programming 2

# Quick Review

**Access**

**Insertion**

**Deletion**

Array/ArrayList

LinkedList

Binary Search  
Tree

Stacks

Queue

|                    | Access      | Insertion                   | Deletion                    |
|--------------------|-------------|-----------------------------|-----------------------------|
| ArrayList          | $O(c)$      | $O(n)$                      | $O(n)$                      |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      |

|                    | Access      | Insertion                   | Deletion                    | Comment                          |
|--------------------|-------------|-----------------------------|-----------------------------|----------------------------------|
| ArrayList          | $O(c)$      | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to front / back   |

what if we want a data structure that holds integers, and has constant time insertion & deletion?

|                    | Access      | Insertion                   | Deletion                    | Comment                          |
|--------------------|-------------|-----------------------------|-----------------------------|----------------------------------|
| Array/ArrayList    | $O(c)$      | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to front / back   |

|                    | <b>Access</b> | <b>Insertion</b>            | <b>Deletion</b>             | <b>Comment</b>                   |
|--------------------|---------------|-----------------------------|-----------------------------|----------------------------------|
| ArrayList          | $O(c)$        | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$        | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$   | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$        | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$        | $O(c)$                      | $O(c)$                      | access limited to front / back   |

what if we want also want constant time access to any item?

# Hashing

A technique for efficiently mapping data elements to indexes in an array so that they can be added, removed and searched quickly →  $O(c)$

- Constant time
  - Insertion
  - Deletion
  - Random access

- Constant time
  - Insertion
  - Deletion
  - Random access
- We know:
  - Possible range of integers is [0...MAX\_INT]
- What is a naïve, brute-force solution?
  - Hint: use an array

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

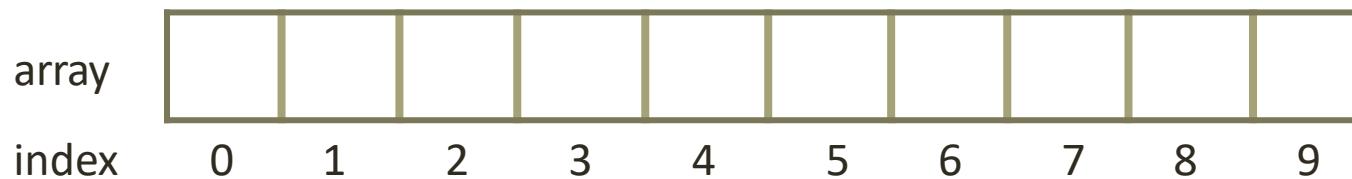
Does this fulfill the constant time insertion,  
deletion, and access requirements?

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

Does this fulfill the constant time insertion,  
deletion, and access requirements?  
Is this realistic ???

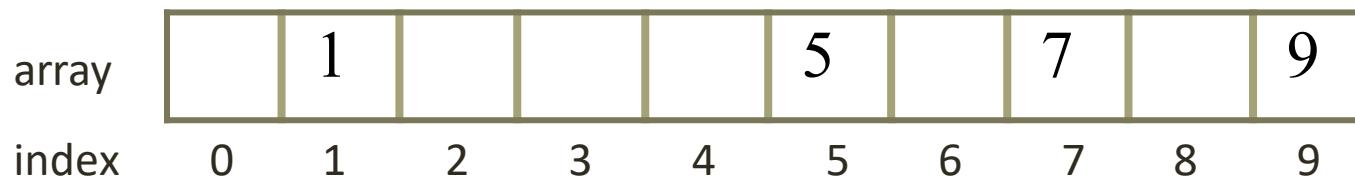
insert:

1, 5, 7, 9



insert:

1, 5, 7, 9



insert:

1, 5, 7, 9, 23?? UH OH

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| array |   | 1 |   |   | 5 |   | 7 |   | 9 |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

insert:

1, 5, 7, 9, 23?? UH OH

- Could make a bigger array, but you could always have a bigger integer

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| array |   | 1 |   |   | 5 |   | 7 |   | 9 |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

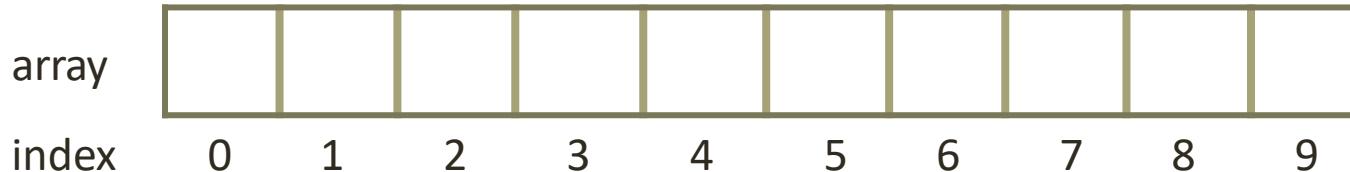
# Mapping to Arrays

- Let's try using a smaller array, and mapping large indices to the range of the smaller array
- Assume range of possible items is [0...99]
  - and assume that we will have <<100 items
- Assume array size is only 10
- How can we make this work for integers?

- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

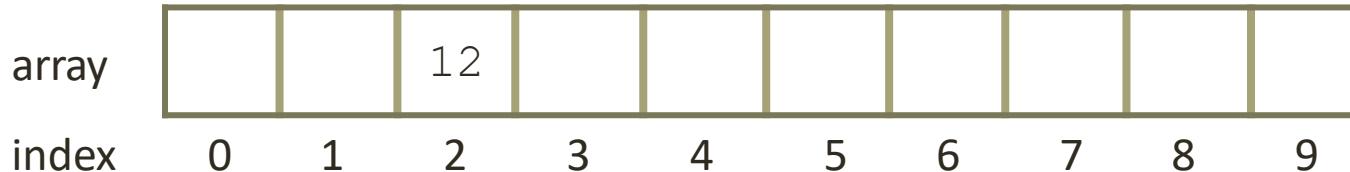
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

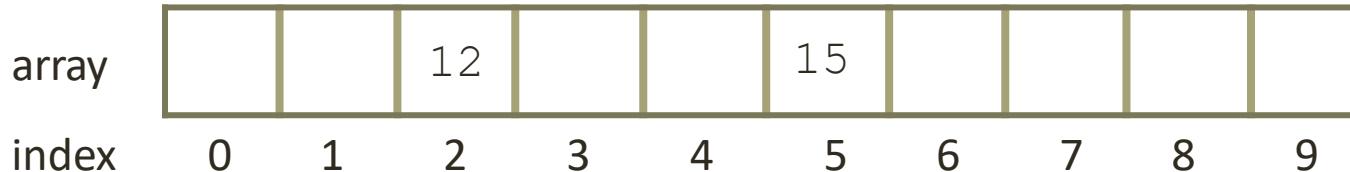
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

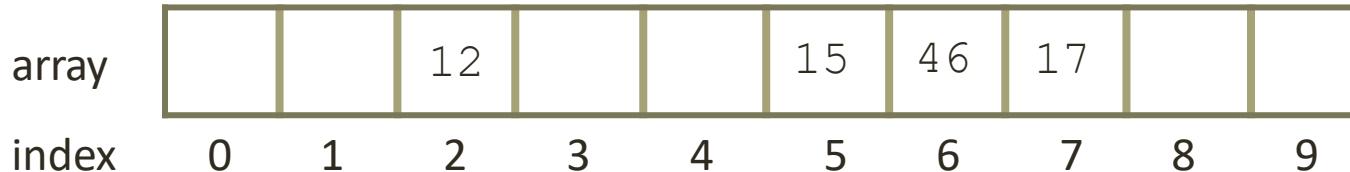
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

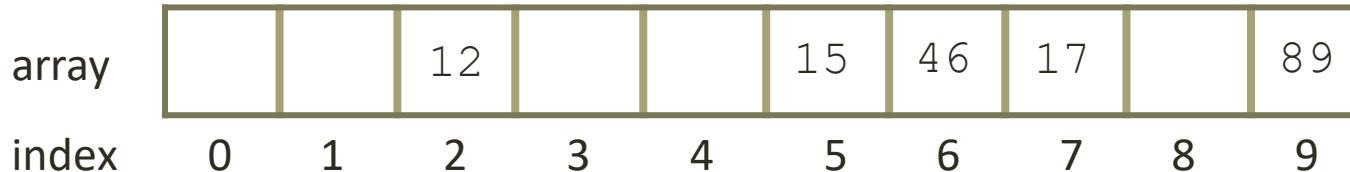
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

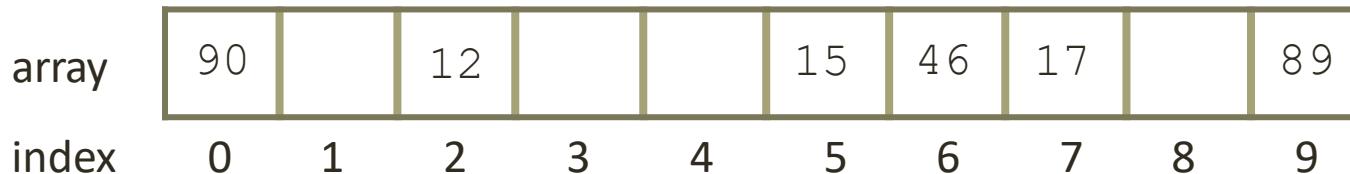
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

12, 15, 17, 46, 89, 90



# Hash table

# Hash table



- A **hash table** is a general storage data structure
  - Also known as a **hash map**
- Insertion, deletion, and access are all  $O(c)$ 
  - Unlike stacks and queues, can access ***all*** items!
  - These are fast!
  - Many applications like databases

|            | Access | Insertion | Deletion | Comment |
|------------|--------|-----------|----------|---------|
| Hash Table | $O(c)$ | $O(c)$    | $O(c)$   | Magic!  |

- A **hash table** is a general storage data structure
  - Also known as a **hash map**
- Insertion, deletion, and access are all  $O(c)$ 
  - Unlike stacks and queues, can access **all** items!
  - These are fast!
  - Many applications like databases

|            | Access | Insertion | Deletion | Comment |
|------------|--------|-----------|----------|---------|
| Hash Table | $O(c)$ | $O(c)$    | $O(c)$   | Magic!  |

No magic in Java, then how?!

# Hash Functions

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

mod operator, %

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

```
int hashFunction(int value)
{
    return Math.abs(value) % array.length;
}
```

- Hash functions will give us some *hash value* which we can map to a valid array index using %
- Empty spots in the array are set to null
- Use any hash value to instantly look-up the index of any item
  - insertion, deletion, and access:  $O(c)$ 
    - *assuming the hash function is  $O(c)$  !*

# Hash Collisions

insert:

**12, 15, 17, 46, 89, 90**

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

`insert:`

`12, 15, 17, 46, 89, 90, 92`

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

insert:

12, 15, 17, 46, 89, 90, 92

↓ collision! where can we put 92?

| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
|-------|----|---|----|---|---|----|----|----|---|----|
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

- Remember: *it is NOT required that two non-equal objects have different hash values*
- Because of this, it is possible for two different objects to return the same hash values
  - This is called a **collision**

insert:

12, 15, 17, 46, 89, 90, 92

↓  
collision! where can we put 92?

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

# Probing

- Looking for another index to use if the preferred index is taken.
- Resolving hash collision by placing elements at other indexes in the table rather than their preferred indexes.
- How?
  - Remember the original goal

# Probing

- Looking for another index to use if the preferred index is taken.
- Resolving hash collision by placing elements at other indexes in the table rather than their preferred indexes.
- How?
  - Remember the original goal

Searching for elements is supposed to be  $O(c)$

# Probing techniques

- Linear probing
- Quadratic probing
- Separate chaining
- And more...

# Insert with linear probing

Collisions are resolved on inserts by  
sequentially scanning the table  
(with wraparound) until an empty  
cell is found

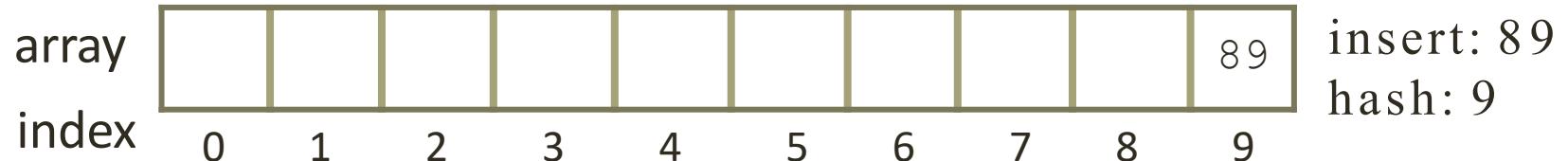
# Linear probing

- When *inserting*
  - If the spot is already taken, simply step forward one index at a time until an empty space is found
  - and, then insert item in empty space
- When *accessing*
  - Start at the hashed value index, and if this is not the item we are searching for, begin stepping forward until the item is found
  - What if we hit the end of the array?
  - When do we stop?

# Linear probing

- When *accessing*
  - Start at the hashed value index, and if this is not the item we are searching for, begin stepping forward until the item is found
  - What if we hit the end of the array? Keep looping
  - When do we stop? At first empty element

# Insert with linear probing



# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

# Insert with linear probing

|       |                                                                                     |            |         |
|-------|-------------------------------------------------------------------------------------|------------|---------|
| array |   | insert: 89 | hash: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |   | insert: 18 | hash: 8 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |   | insert: 49 | hash: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |  | insert: 58 | hash: 8 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |

# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                     |            |
|-------|-------------------------------------------------------------------------------------|------------|
| array |  | insert: 58 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 | hash: 8    |

|       |                                                                                      |           |
|-------|--------------------------------------------------------------------------------------|-----------|
| array |  | insert: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                  | hash: 9   |

# Access with linear probing

- If the table is not full, the item we seek, or an empty cell, will eventually be found
- Cost?
  - Recall that we are hoping for  $O(c)$
- Access operation follows the same path as insert...
  - if empty cell Reached, item not found
- How do we find 58?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 58 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 8    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Why?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Because items serve as placeholders during collision resolution

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Because items serve as placeholders during collision resolution

| array | 49 | 58 | 9 |   |   |   |   | 18 |   | search: 89<br>hash: 9 |
|-------|----|----|---|---|---|---|---|----|---|-----------------------|
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8 | 9                     |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

How do we find 9?

| array | 49 | 58 | 9 |   |   |   |   |   |   | search: 89 |
|-------|----|----|---|---|---|---|---|---|---|------------|
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

How do we find 9?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

- We must use **lazy deletion**, which marks items as deleted rather than actually removing them

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

| How do we find 9? |    |    |   |   |   |   |   |    |    |            |
|-------------------|----|----|---|---|---|---|---|----|----|------------|
| array             | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index             | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |
| deleted           | F  | F  | F |   |   |   |   |    | F  | T          |

- We must use **lazy deletion**, which marks items as deleted rather than actually removing them

# Performance

If no collisions occur, then the performance of insert, delete, and access is \_\_\_\_\_

# Performance

If no collisions occur, then the performance of insert, delete, and access is **O (c)**

## Another example

|       |                                                                                                                                                                                                                                                                    |    |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|-----------------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25<br>hash: 5 |
|       | 1                                                                                                                                                                                                                                                                  | 91 | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
| index | 0                                                                                                                                                                                                                                                                  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |                       |

# Another example

|       |                                                                                                                                                                                                                                                                    |    |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|-----------------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25<br>hash: 5 |
|       | 1                                                                                                                                                                                                                                                                  | 91 | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
| index | 0                                                                                                                                                                                                                                                                  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |                       |

|       |                                                                                                                                                                                                                                                                      |    |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|----|---|-------|---|---|---|---|---|---|---|---|---|---|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td>25</td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 | 25 | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|       | 1                                                                                                                                                                                                                                                                    | 91 | 71 |    | 5  | 45 | 7 | 25 | 9 |    |   |       |   |   |   |   |   |   |   |   |   |   |
| index | 0                                                                                                                                                                                                                                                                    | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9  |   |       |   |   |   |   |   |   |   |   |   |   |

# Another example

|       |                                                                                                                                                                                                                                                                    |         |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |            |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |         | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25 |
|       | 1                                                                                                                                                                                                                                                                  | 91      | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index | 0                                                                                                                                                                                                                                                                  | 1       | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index |                                                                                                                                                                                                                                                                    | hash: 5 |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |            |

|       |                                                                                                                                                                                                                                                                      |         |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |            |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|----|----|----|----|---|----|---|----|---|-------|---|---|---|---|---|---|---|---|---|---|------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td>25</td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |         | 1  | 91 | 71 |    | 5 | 45 | 7 | 25 | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | search: 95 |
|       | 1                                                                                                                                                                                                                                                                    | 91      | 71 |    | 5  | 45 | 7 | 25 | 9 |    |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index | 0                                                                                                                                                                                                                                                                    | 1       | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9  |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index |                                                                                                                                                                                                                                                                      | hash: 5 |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |            |

How many indices will be checked?

# Clustering

- If an item's natural spot is taken, it goes in the next open spot, making a cluster for that hash
  - **Clustering** happens because once there is a collision, there is a high probability more will occur
  - Thus, any item that hashes into a cluster will require several attempts to resolve the collision

| array | 0 | 1 | 2  | 3  | 4 | 5 | 6  | 7 | 8  | 9 |
|-------|---|---|----|----|---|---|----|---|----|---|
| index |   | 1 | 91 | 71 |   | 5 | 45 | 7 | 25 | 9 |

Let  $h(k)$  be a [hash function](#) that maps an element  $k$  to an integer in  $[0, m-1]$ , where  $m$  is the size of the table. Let the  $i^{\text{th}}$  probe position for a value  $k$  be given by the function

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \pmod{m}$$

where  $c_2 \neq 0$ . If  $c_2 = 0$ , then  $h(k, i)$  degrades to a [linear probe](#). For a given [hash table](#), the values of  $c_1$  and  $c_2$  remain constant.

# Quadratic probing

- Probing is how to resolve collisions
  - to address clustering, we will use **quadratic probing**
- If  $\text{hash}(\text{key}) = H$ , and the cell at index  $H$  is occupied:
  - Try  $(H+1^2) \bmod \text{size}$
  - Then  $(H+2^2) \bmod \text{size}$
  - Then  $(H+3^2) \bmod \text{size}$
  - and so on...
  - Wrap around to beginning of array if necessary

# Quadratic Probing...

The screenshot shows a web browser window titled "Hashing Quadratic Probing Ani". The URL is [www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html](http://www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html). The page content is titled "Hashing Using Quadratic Probing Animation by Y. Daniel Liang". A usage instruction is provided: "Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold factor and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99." Below this, a table shows the current state of a hash table of size 11. The table has 11 slots, indexed [0] to [10]. Slots [0], [4], [5], [6], [7], [8], [9], and [10] are empty. Slots [1], [2], [3], and [11] contain the values 44, 4, 16, and 28 respectively. At the bottom, there are input fields for "Enter Initial Table Size" (set to 11) and "Enter a Load Factor Threshold" (set to 0.75). Below these are buttons for "Enter a key:", "Search", "Insert", "Remove", and "Remove All".

Source:

<http://www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html>

(<http://www.cs.armstrong.edu/liang/animation/web/LinearProbing.html>)

# Concerns...

- Is quadratic probing guaranteed to find an open spot?  
can it search the same spot twice?
- Suppose the array size is 16, and hash (key) = 0

$$0 \% 16 = 0$$

$$(0+1^2) \% 16 = 1$$

$$(0+2^2) \% 16 = 4$$

$$(0+3^2) \% 16 = 9$$

$$(0+4^2) \% 16 = 0$$

$$(0+5^2) \% 16 = 9$$

$$(0+6^2) \% 16 = 4$$

$$(0+7^2) \% 16 = 1$$

## Concerns...

- Is quadratic probing guaranteed to find an open spot?  
can it search the same spot twice?
- Suppose the array size is 16, and hash (key) = 0

$$0 \% 16 = 0$$

$$(0+1^2) \% 16 = 1$$

$$(0+2^2) \% 16 = 4$$

$$(0+3^2) \% 16 = 9$$

$$(0+4^2) \% 16 = 0$$

$$(0+5^2) \% 16 = 9$$

$$(0+6^2) \% 16 = 4$$

$$(0+7^2) \% 16 = 1$$

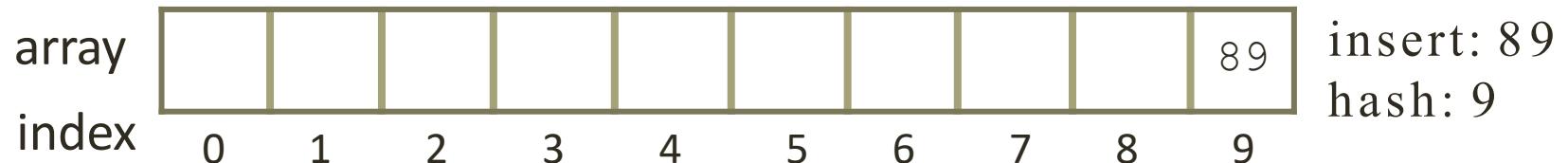
**limitation:** at most, half of the table can be used to resolve collisions

once table is half full it is difficult to find an empty spot

...Called **secondary clustering**

# Insert with quadratic probing

# Insert with quadratic probing

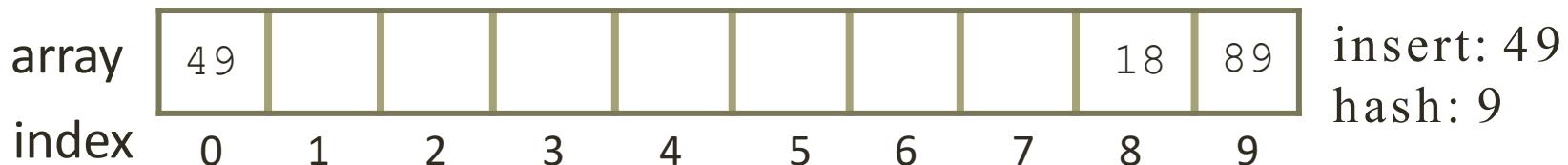
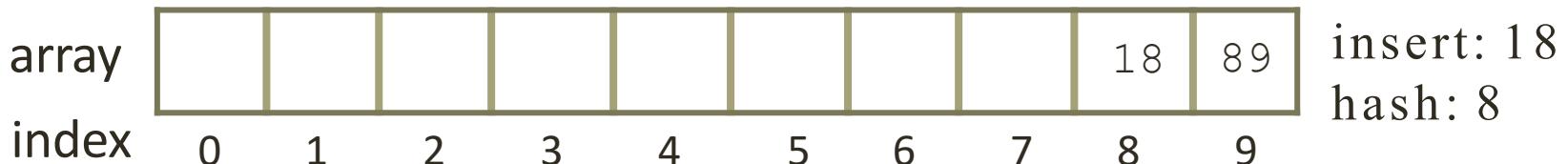
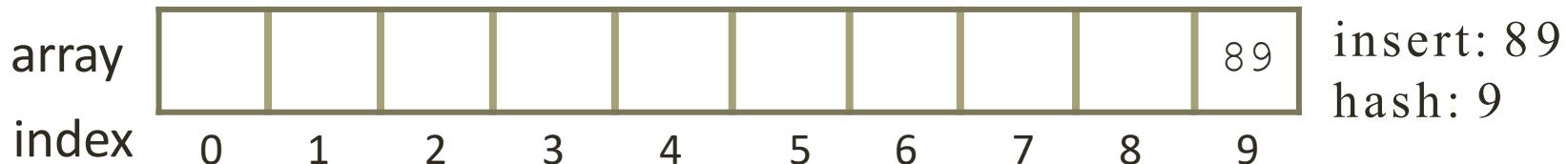


# Insert with quadratic probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

# Insert with quadratic probing



# Insert with quadratic probing

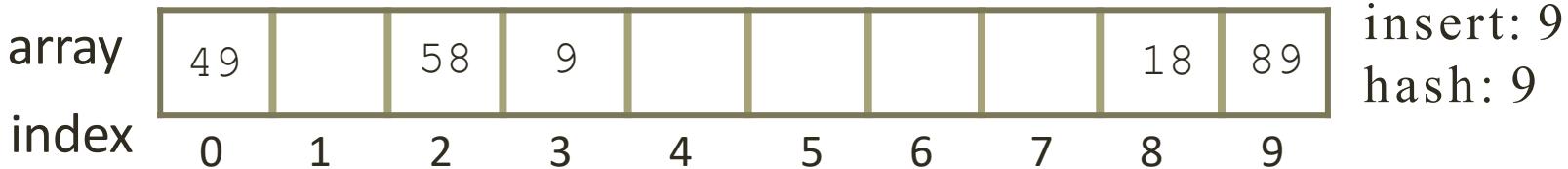
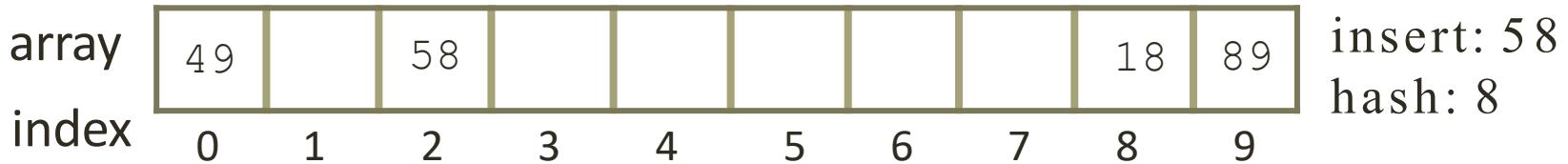
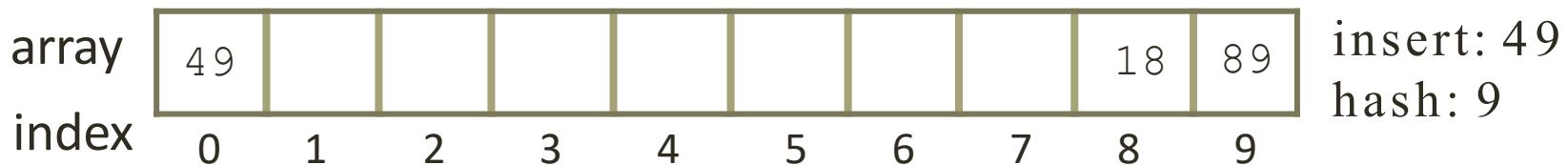
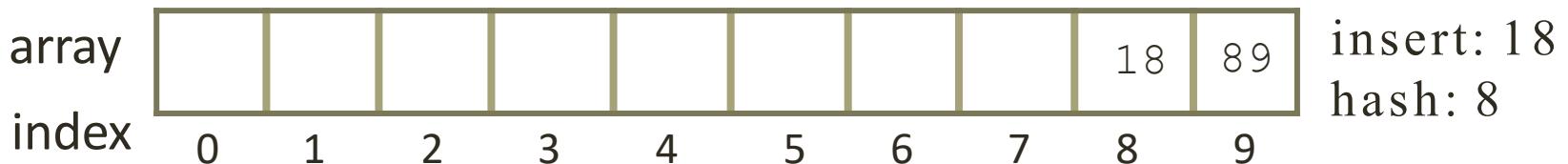
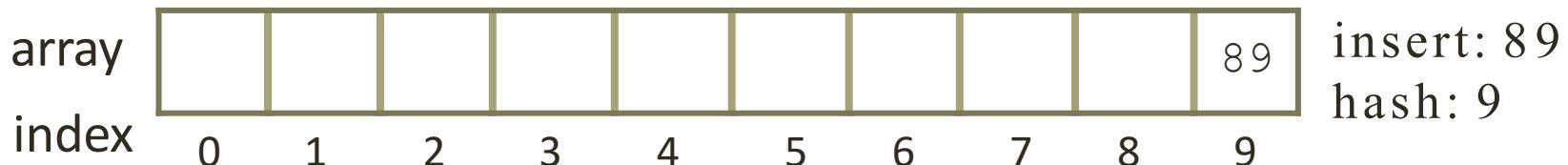
|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                     |            |
|-------|-------------------------------------------------------------------------------------|------------|
| array |  | insert: 58 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 | hash: 8    |

# Insert with quadratic probing



|       |                                                                                                                                                          |    |    |   |    |    |    |    |    |    |    |    |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|---|----|----|----|----|----|----|----|----|
| array | <table border="1"> <tr> <td>77</td><td>11</td><td></td><td>33</td><td>4</td><td>16</td><td>28</td><td></td><td></td><td>22</td><td>21</td></tr> </table> | 77 | 11 |   | 33 | 4  | 16 | 28 |    |    | 22 | 21 |
| 77    | 11                                                                                                                                                       |    | 33 | 4 | 16 | 28 |    |    | 22 | 21 |    |    |
| index | 0    1    2    3    4    5    6    7    8    9    10                                                                                                     |    |    |   |    |    |    |    |    |    |    |    |

- Try  $(H+1^2) \bmod \text{size}$
  - Then  $(H+2^2) \bmod \text{size}$
  - Then  $(H+3^2) \bmod \text{size}$
- insert: 55  
hash: 0

Can quadratic probing find a spot?!

...solution!

Rehashing

# ...solution!

- Resize the table (array)
  - Just like resizing an array, we resize the table to be larger
    - Usually a larger prime number
  - Instead of a simple copy-everything-over, all items must be rehashed
    - why?

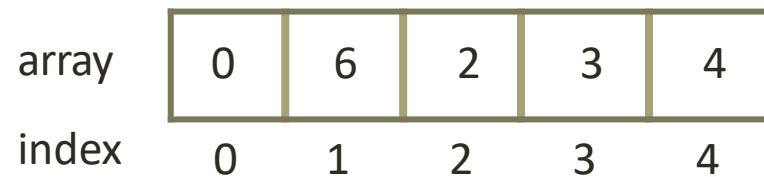
# ...solution!

- Resize the table (array)
  - Just like resizing an array, we resize the table to be larger
    - Usually a larger prime number
  - Instead of a simple copy-everything-over, all items must be rehashed
    - why?

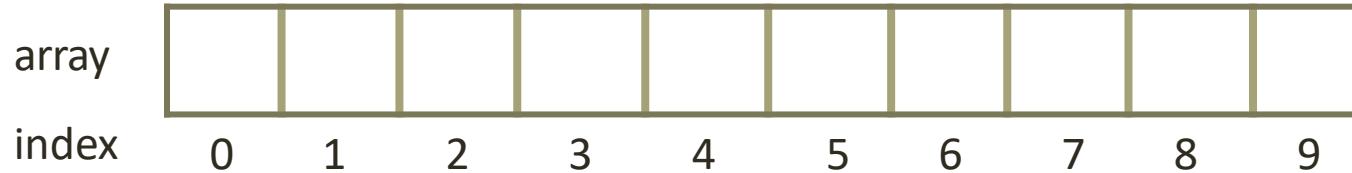
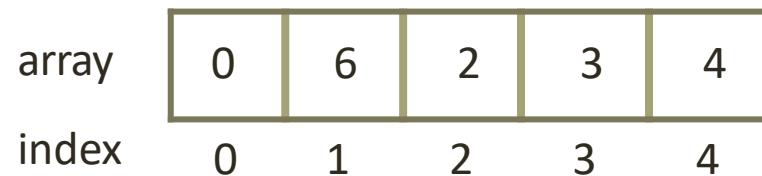
# Rehashing

- Resizing a hash table to increase its capacity and enabling it to store more elements, or store them more efficiently

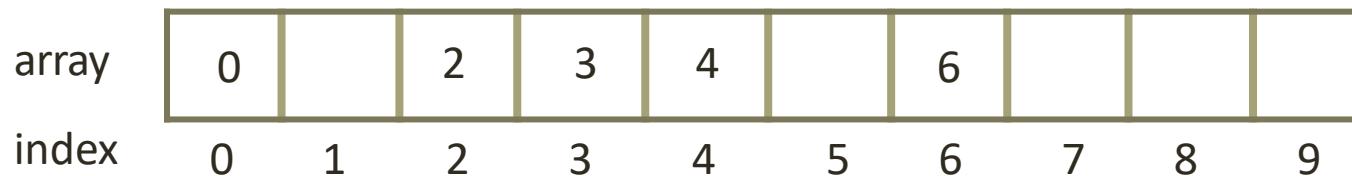
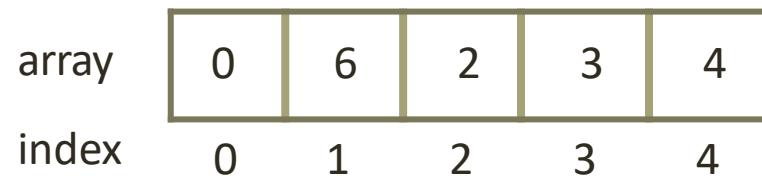
# Rehashing example



# Rehashing example



# Rehashing example



# Load factor

- The measure of how full the hash table is
- Table size relative to the array capacity
  - the fraction of the table that is full
    - Number of items in array / size of array
    - Called  $\lambda$
    - $0 \leq \lambda \leq 1$
- Instead of rehashing when the table is half full, rehash when  $\lambda$  becomes large
- Standard for Java Hashmap is .75

# Rehashing

- Resizing a hash table to increase its capacity and enabling it to store more elements, or store them more efficiently

Can you think of an alternative for collision management?

# Separate chaining

- Why not make each spot in the array capable of holding more than one item?

# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array

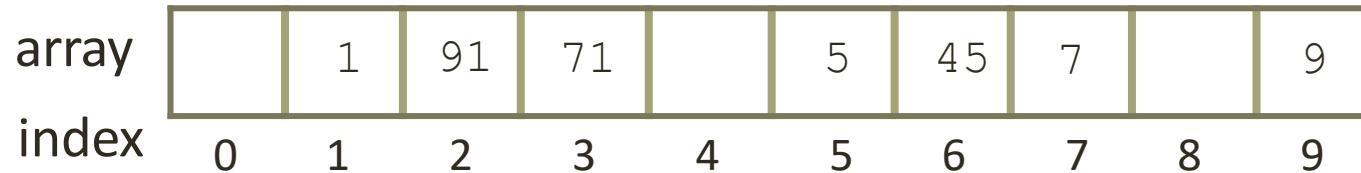
# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array
- For insertion, append the item to the end of the list
  - Insertion is  $O(c)$  if we have what?

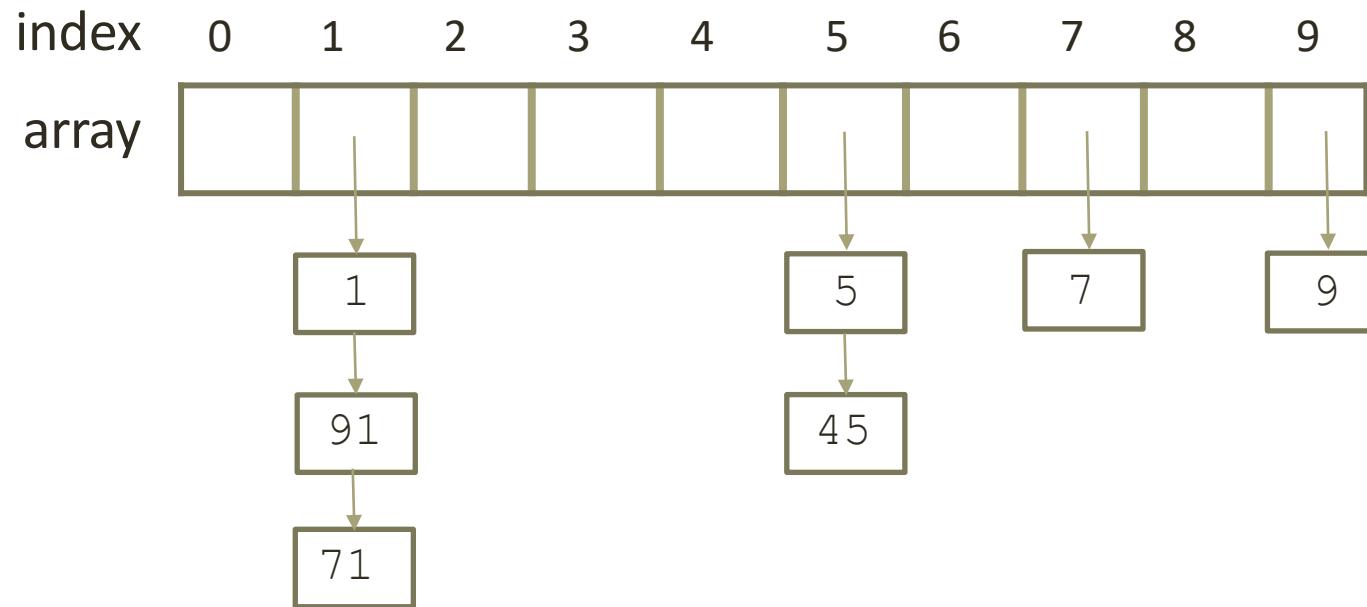
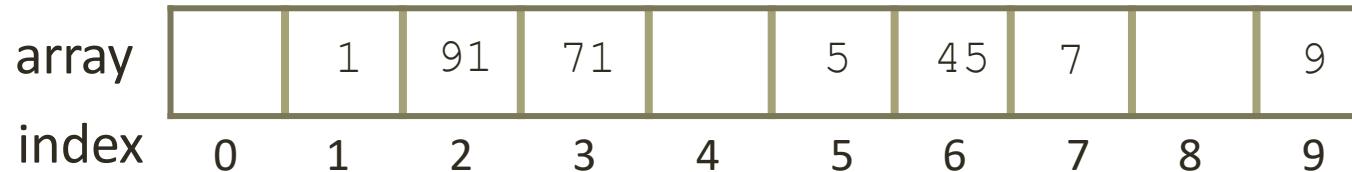
# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array
- For insertion, append the item to the end of the list
  - Insertion is  $O(c)$  if we have what?
- Accessing is a linear scan through the list
  - Fast if the list is short

# Separate chaining



# Separate chaining



What data structure we can use for implementation?

# Performance

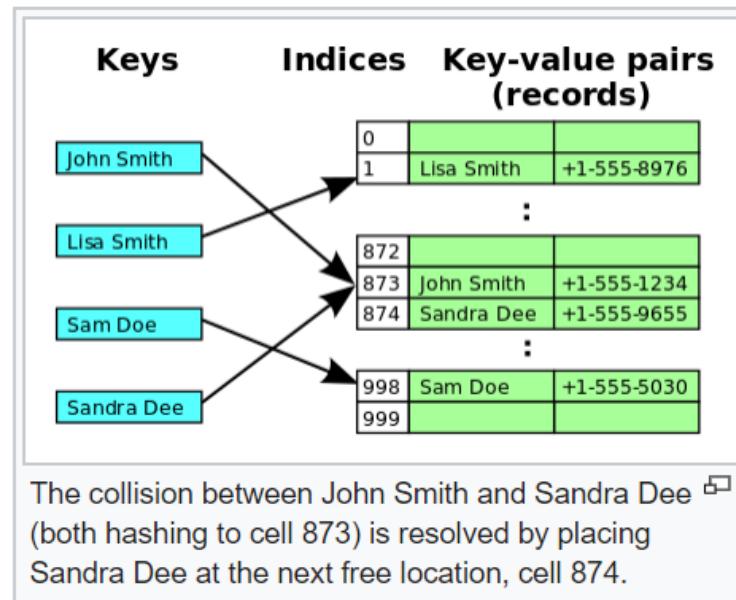
- Clusters can still happen with separate chaining
- When to resize?
  - One easy rule
    - resize the hash table once its size exceed its capacity.
  - However, most hash table implementation resize before that
    - How?

# Load factor

- The measure of how full the hash table is
- Table size relative to the array capacity
  - the fraction of the table that is full
  - Called  $\lambda$
  - $0 \leq \lambda \leq 1$
- Instead of rehashing when the table is half full, rehash when  $\lambda$  becomes large

# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?



# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?
- One solution is to somehow generate an integer from a string
  - Length of string?
  - Sum of all characters?
  - Some combination of both?

# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?
- One solution is to somehow generate an integer from a string
  - Length of string?
  - Sum of all characters?
  - Some combination of both?
- A method that generates an integer index given any object is called a ***hash function***

A couple of rules...

- Always returns the same number for the same object

## A couple of rules...

- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects

# A couple of rules...

- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects
- Good hash functions return evenly distributed numbers for the input items

## A couple of rules...

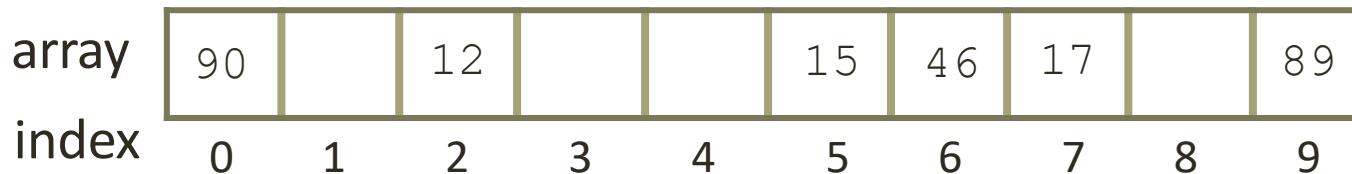
- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects
- Good hash functions return evenly distributed numbers for the input items
- *It is not required that two non-equal objects have different hash values*

# Java's hashCode

- Every Object in Java has a method hashCode
- Returns an integer based on the object
- Default for this method (if you don't override it) is to return the memory address of the object
- Will not be very well-distributed if your items are contiguous in memory

A bit more on hash functions...

- ints have an obvious hash value



- What about Strings? Books? Shapes?...
- We must not overlook the requirement of a *good* hash functions

# Remember...

- Hash functions take any item as input and produce an integer as output
- Given the same input the function always returns the same output
- Two different inputs MAY have the same hash value

# Thinking about **chars** and **Strings**

- ASCII defines an encoding for characters
  - 'a' = 97
  - 'b' = 98
  - ...
  - 'z' = 122
  - '2' = 50 - ...
- The `char` type is actually just a small integer
  - 8 bits (if using ASCII, 16 bits if using UTF-16 coding for the `char`) instead of the usual 32

| Dec | Hx | Oct | Char       |                          | Dec | Hx | Oct | Html  | Chr          |  | Dec | Hx | Oct | Html  | Chr        |
|-----|----|-----|------------|--------------------------|-----|----|-----|-------|--------------|--|-----|----|-----|-------|------------|
| 0   | 0  | 000 | <b>NUL</b> | (null)                   | 32  | 20 | 040 | &#32; | <b>Space</b> |  | 64  | 40 | 100 | &#64; | <b>Ø</b>   |
| 1   | 1  | 001 | <b>SOH</b> | (start of heading)       | 33  | 21 | 041 | &#33; | <b>!</b>     |  | 65  | 41 | 101 | &#65; | <b>A</b>   |
| 2   | 2  | 002 | <b>STX</b> | (start of text)          | 34  | 22 | 042 | &#34; | <b>"</b>     |  | 66  | 42 | 102 | &#66; | <b>B</b>   |
| 3   | 3  | 003 | <b>ETX</b> | (end of text)            | 35  | 23 | 043 | &#35; | <b>#</b>     |  | 67  | 43 | 103 | &#67; | <b>C</b>   |
| 4   | 4  | 004 | <b>EOT</b> | (end of transmission)    | 36  | 24 | 044 | &#36; | <b>\$</b>    |  | 68  | 44 | 104 | &#68; | <b>D</b>   |
| 5   | 5  | 005 | <b>ENQ</b> | (enquiry)                | 37  | 25 | 045 | &#37; | <b>%</b>     |  | 69  | 45 | 105 | &#69; | <b>E</b>   |
| 6   | 6  | 006 | <b>ACK</b> | (acknowledge)            | 38  | 26 | 046 | &#38; | <b>&amp;</b> |  | 70  | 46 | 106 | &#70; | <b>F</b>   |
| 7   | 7  | 007 | <b>BEL</b> | (bell)                   | 39  | 27 | 047 | &#39; | <b>'</b>     |  | 71  | 47 | 107 | &#71; | <b>G</b>   |
| 8   | 8  | 010 | <b>BS</b>  | (backspace)              | 40  | 28 | 050 | &#40; | <b>(</b>     |  | 72  | 48 | 110 | &#72; | <b>H</b>   |
| 9   | 9  | 011 | <b>TAB</b> | (horizontal tab)         | 41  | 29 | 051 | &#41; | <b>)</b>     |  | 73  | 49 | 111 | &#73; | <b>I</b>   |
| 10  | A  | 012 | <b>LF</b>  | (NL line feed, new line) | 42  | 2A | 052 | &#42; | <b>*</b>     |  | 74  | 4A | 112 | &#74; | <b>J</b>   |
| 11  | B  | 013 | <b>VT</b>  | (vertical tab)           | 43  | 2B | 053 | &#43; | <b>+</b>     |  | 75  | 4B | 113 | &#75; | <b>K</b>   |
| 12  | C  | 014 | <b>FF</b>  | (NP form feed, new page) | 44  | 2C | 054 | &#44; | <b>,</b>     |  | 76  | 4C | 114 | &#76; | <b>L</b>   |
| 13  | D  | 015 | <b>CR</b>  | (carriage return)        | 45  | 2D | 055 | &#45; | <b>-</b>     |  | 77  | 4D | 115 | &#77; | <b>M</b>   |
| 14  | E  | 016 | <b>SO</b>  | (shift out)              | 46  | 2E | 056 | &#46; | <b>.</b>     |  | 78  | 4E | 116 | &#78; | <b>N</b>   |
| 15  | F  | 017 | <b>SI</b>  | (shift in)               | 47  | 2F | 057 | &#47; | <b>/</b>     |  | 79  | 4F | 117 | &#79; | <b>O</b>   |
| 16  | 10 | 020 | <b>DLE</b> | (data link escape)       | 48  | 30 | 060 | &#48; | <b>Ø</b>     |  | 80  | 50 | 120 | &#80; | <b>P</b>   |
| 17  | 11 | 021 | <b>DC1</b> | (device control 1)       | 49  | 31 | 061 | &#49; | <b>1</b>     |  | 81  | 51 | 121 | &#81; | <b>Q</b>   |
| 18  | 12 | 022 | <b>DC2</b> | (device control 2)       | 50  | 32 | 062 | &#50; | <b>2</b>     |  | 82  | 52 | 122 | &#82; | <b>R</b>   |
| 19  | 13 | 023 | <b>DC3</b> | (device control 3)       | 51  | 33 | 063 | &#51; | <b>3</b>     |  | 83  | 53 | 123 | &#83; | <b>S</b>   |
| 20  | 14 | 024 | <b>DC4</b> | (device control 4)       | 52  | 34 | 064 | &#52; | <b>4</b>     |  | 84  | 54 | 124 | &#84; | <b>T</b>   |
| 21  | 15 | 025 | <b>NAK</b> | (negative acknowledge)   | 53  | 35 | 065 | &#53; | <b>5</b>     |  | 85  | 55 | 125 | &#85; | <b>U</b>   |
| 22  | 16 | 026 | <b>SYN</b> | (synchronous idle)       | 54  | 36 | 066 | &#54; | <b>6</b>     |  | 86  | 56 | 126 | &#86; | <b>V</b>   |
| 23  | 17 | 027 | <b>ETB</b> | (end of trans. block)    | 55  | 37 | 067 | &#55; | <b>7</b>     |  | 87  | 57 | 127 | &#87; | <b>W</b>   |
| 24  | 18 | 030 | <b>CAN</b> | (cancel)                 | 56  | 38 | 070 | &#56; | <b>8</b>     |  | 88  | 58 | 130 | &#88; | <b>X</b>   |
| 25  | 19 | 031 | <b>EM</b>  | (end of medium)          | 57  | 39 | 071 | &#57; | <b>9</b>     |  | 89  | 59 | 131 | &#89; | <b>Y</b>   |
| 26  | 1A | 032 | <b>SUB</b> | (substitute)             | 58  | 3A | 072 | &#58; | <b>:</b>     |  | 90  | 5A | 132 | &#90; | <b>Z</b>   |
| 27  | 1B | 033 | <b>ESC</b> | (escape)                 | 59  | 3B | 073 | &#59; | <b>:</b>     |  | 91  | 5B | 133 | &#91; | <b>[</b>   |
| 28  | 1C | 034 | <b>FS</b>  | (file separator)         | 60  | 3C | 074 | &#60; | <b>&lt;</b>  |  | 92  | 5C | 134 | &#92; | <b>\</b>   |
| 29  | 1D | 035 | <b>GS</b>  | (group separator)        | 61  | 3D | 075 | &#61; | <b>=</b>     |  | 93  | 5D | 135 | &#93; | <b>]</b>   |
| 30  | 1E | 036 | <b>RS</b>  | (record separator)       | 62  | 3E | 076 | &#62; | <b>&gt;</b>  |  | 94  | 5E | 136 | &#94; | <b>^</b>   |
| 31  | 1F | 037 | <b>US</b>  | (unit separator)         | 63  | 3F | 077 | &#63; | <b>?</b>     |  | 95  | 5F | 137 | &#95; | <b>_</b>   |
|     |    |     |            |                          |     |    |     |       |              |  |     |    |     |       | <b>DEL</b> |

- A String is essentially an array of char

```
String s = "hello";
```



- Java hides these details
- How can we use this to create a hash function for Strings?

- A String is essentially an array of char

```
String s = "hello";
```



- Java hides these details
- How can we use this to create a hash function for Strings?
  - How about we add all the values?

- A String is essentially an array of char

```
String s = "hello";
```

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 104 | 101 | 108 | 108 | 111 |
|-----|-----|-----|-----|-----|

- Java hides these details
- How can we use this to create a hash function for Strings?
  - How about we add all the values? → Not great!
  - Why?
  - It's actually more like this:
    - $h(s) = s[0]*31^{n-1} + s[1]*31^{n-2} + \dots + s[n-1]$

The good and widely used way to define the hash of a string  $s$  of length  $n$  is

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \dots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

where  $p$  and  $m$  are some chosen, positive numbers. It is called a **polynomial rolling hash function**.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems
- Quadratic probing
  - Has lesser clustering problems
  - Requires  $\lambda < 0.5$ , and prime table size

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems
- Quadratic probing
  - Has lesser clustering problems
  - Requires  $\lambda < 0.5$ , and prime table size
- Separate chaining
  - Probably the easiest to implement, as well as the best performing

**What is the load factor  $\lambda$  for the following hash table?**

- A) 4
- B) 6
- C) 0.4
- D) 0.5
- E) 0.6

| array | 104 | 34 |   | 19 | 111 | 98 |   | 52 |   |   |
|-------|-----|----|---|----|-----|----|---|----|---|---|
| index | 0   | 1  | 2 | 3  | 4   | 5  | 6 | 7  | 8 | 9 |

## Using linear probing, in what index will item 93 be added?

- A) 1
- B) 5
- C) 6
- D) 7

|       |    |   |   |    |    |   |   |    |    |
|-------|----|---|---|----|----|---|---|----|----|
| array | 49 |   | 9 | 58 | 34 |   |   | 18 | 89 |
| index | 0  | 1 | 2 | 3  | 4  | 5 | 6 | 7  | 8  |

**Using linear probing, in what index will item 22 be added?**

- A) 1
- B) 5
- C) 6
- D) 7

|       |    |   |   |    |    |   |   |    |    |
|-------|----|---|---|----|----|---|---|----|----|
| array | 49 |   | 9 | 58 | 34 |   |   | 18 | 89 |
| index | 0  | 1 | 2 | 3  | 4  | 5 | 6 | 7  | 8  |

# Recap

- Hash tables
  - collection structure with  $O(c)$  for major operations
- but!...
  - Hash function must minimize collisions
    - *Should evenly distribute values across all possible integers*
  - Collisions must be carefully dealt with
  - Hash function runtime must be fast
- No ordering
  - *How do we find the smallest item in a hash table?*
  - *In a BST?*

Hash Tables  
CSC220 | Computer Programming 2

# Quick Review

**Access**

**Insertion**

**Deletion**

Array/ArrayList

LinkedList

Binary Search  
Tree

Stacks

Queue

|                    | Access      | Insertion                   | Deletion                    |
|--------------------|-------------|-----------------------------|-----------------------------|
| ArrayList          | $O(c)$      | $O(n)$                      | $O(n)$                      |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      |

|                    | Access      | Insertion                   | Deletion                    | Comment                          |
|--------------------|-------------|-----------------------------|-----------------------------|----------------------------------|
| Array/ArrayList    | $O(c)$      | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to front / back   |

what if we want a data structure that holds integers, and has constant time insertion & deletion?

|                    | Access      | Insertion                   | Deletion                    | Comment                          |
|--------------------|-------------|-----------------------------|-----------------------------|----------------------------------|
| Array/ArrayList    | $O(c)$      | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$      | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$      | $O(c)$                      | $O(c)$                      | access limited to front / back   |

|                    | <b>Access</b> | <b>Insertion</b>            | <b>Deletion</b>             | <b>Comment</b>                   |
|--------------------|---------------|-----------------------------|-----------------------------|----------------------------------|
| Array/ArrayList    | $O(c)$        | $O(n)$                      | $O(n)$                      | must know size ahead of time     |
| LinkedList         | $O(n)$        | $O(n)$ or<br>$O(c)$ on ends | $O(n)$ or<br>$O(c)$ on ends | can allocate new items on demand |
| Binary Search Tree | $O(\log n)$   | $O(\log n)$                 | $O(\log n)$                 | must be balanced                 |
| Stacks             | $O(c)$        | $O(c)$                      | $O(c)$                      | access limited to top            |
| Queue              | $O(c)$        | $O(c)$                      | $O(c)$                      | access limited to front / back   |

what if we want also want constant time access to any item?

# Hashing

A technique for efficiently mapping data elements to indexes in an array so that they can be added, removed and searched quickly →  $O(c)$

- Constant time
  - Insertion
  - Deletion
  - Random access

- Constant time
  - Insertion
  - Deletion
  - Random access
- We know:
  - Possible range of integers is [0...MAX\_INT]
- What is a naïve, brute-force solution?
  - Hint: use an array

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

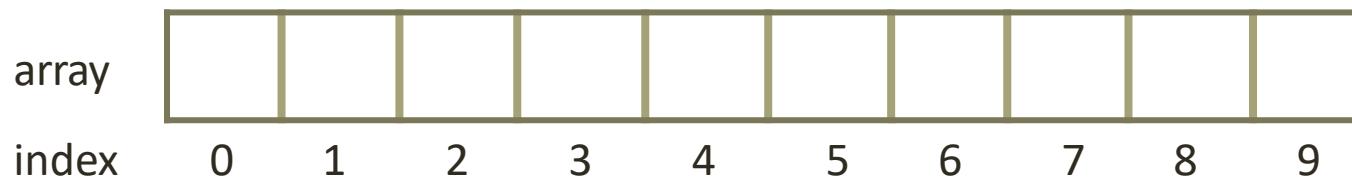
Does this fulfill the constant time insertion,  
deletion, and access requirements?

- Create a gigantic array of size MAX\_INT
- Initialize everything to -1
- When inserting a number n, put it in the array at index n
- When accessing a number n, check if index n is equal to -1 or not
- When deleting a number n, set array at index n to -1

Does this fulfill the constant time insertion,  
deletion, and access requirements?  
Is this realistic ???

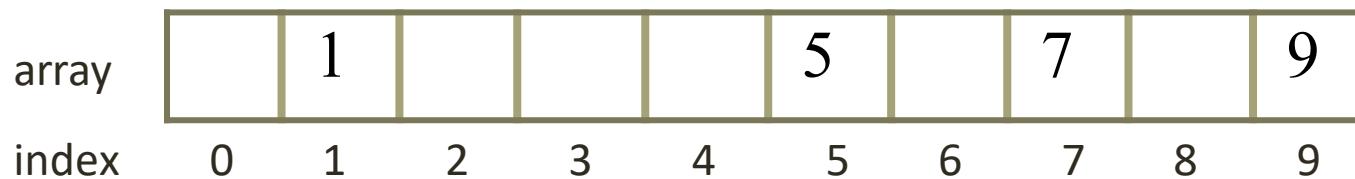
insert:

1, 5, 7, 9



insert:

1, 5, 7, 9



insert:

1, 5, 7, 9, 23?? UH OH

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| array |   | 1 |   |   | 5 |   | 7 |   | 9 |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

insert:

1, 5, 7, 9, 23?? UH OH

- Could make a bigger array, but you could always have a bigger integer

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
| array |   | 1 |   |   | 5 |   | 7 |   | 9 |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

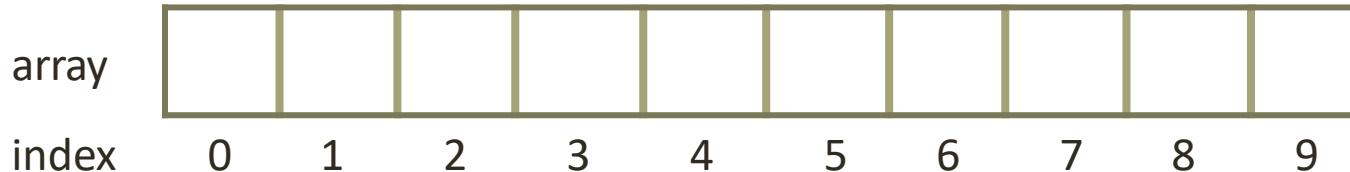
# Mapping to Arrays

- Let's try using a smaller array, and mapping large indices to the range of the smaller array
- Assume range of possible items is [0...99]
  - and assume that we will have <<100 items
- Assume array size is only 10
- How can we make this work for integers?

- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

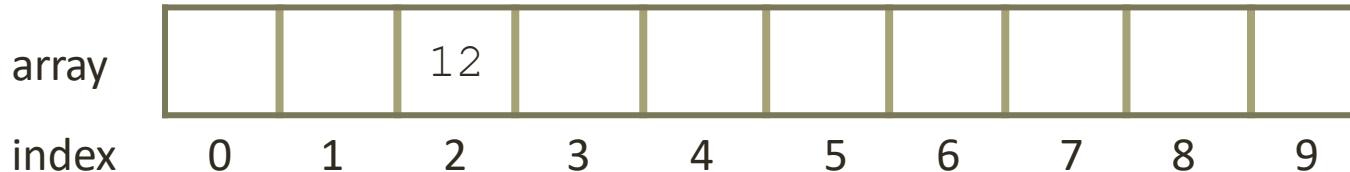
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

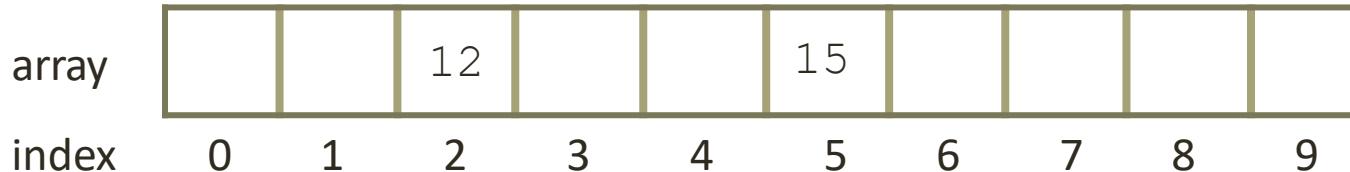
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

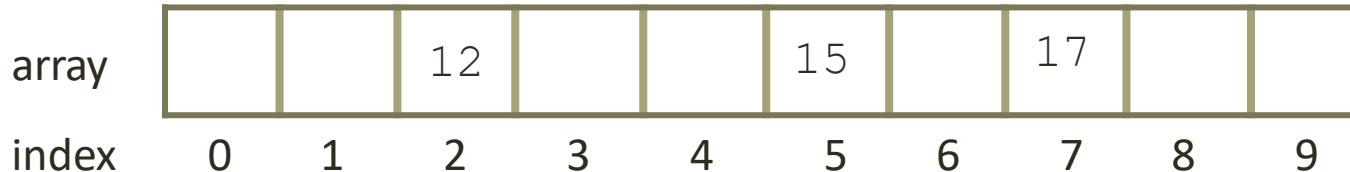
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

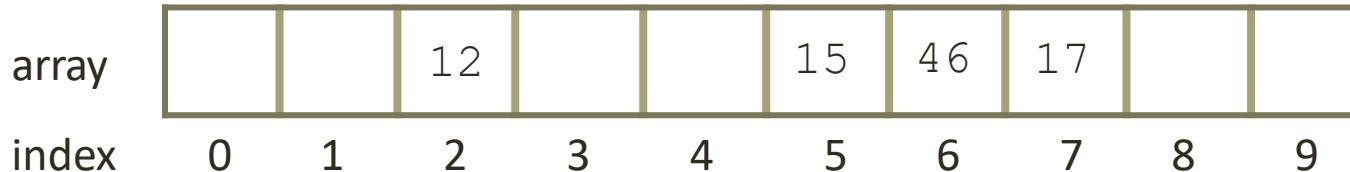
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

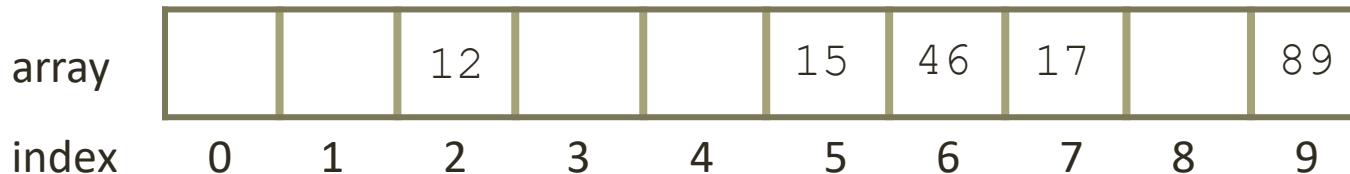
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

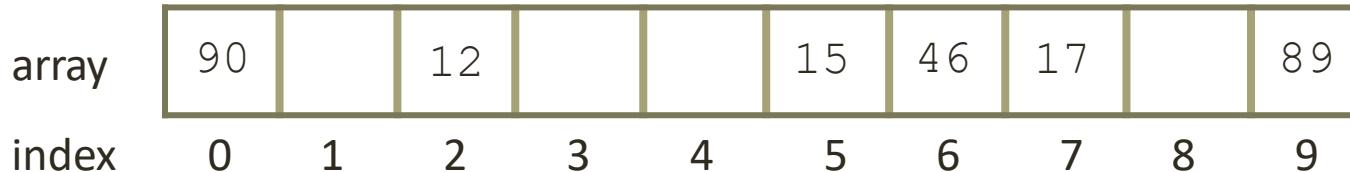
12, 15, 17, 46, 89, 90



- Use the mod operator,  $\%$ 
  - Guarantee to return a number in the range  $[0 \dots (\text{size} - 1)]$
- Mod the input value by the array size for new index

insert:

12, 15, 17, 46, 89, 90



# Hash table

# Hash table



- A **hash table** is a general storage data structure
  - Also known as a **hash map**
- Insertion, deletion, and access are all  $O(c)$ 
  - Unlike stacks and queues, can access ***all*** items!
  - These are fast!
  - Many applications like databases

|            | Access | Insertion | Deletion | Comment |
|------------|--------|-----------|----------|---------|
| Hash Table | $O(c)$ | $O(c)$    | $O(c)$   | Magic!  |

- A **hash table** is a general storage data structure
  - Also known as a **hash map**
- Insertion, deletion, and access are all  $O(c)$ 
  - Unlike stacks and queues, can access **all** items!
  - These are fast!
  - Many applications like databases

|            | Access | Insertion | Deletion | Comment |
|------------|--------|-----------|----------|---------|
| Hash Table | $O(c)$ | $O(c)$    | $O(c)$   | Magic!  |

No magic in Java, then how?!

# Hash Functions

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

mod operator, %

- A method for rapidly mapping between element values and preferred array indexes at which to store those value.
  - Like what?!

```
int hashFunction(int value)
{
    return Math.abs(value) % array.length;
}
```

- Hash functions will give us some *hash value* which we can map to a valid array index using %
- Empty spots in the array are set to null
- Use any hash value to instantly look-up the index of any item
  - insertion, deletion, and access:  $O(c)$ 
    - *assuming the hash function is  $O(c)$  !*

# Hash Collisions

insert:

**12, 15, 17, 46, 89, 90**

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

`insert:`

`12, 15, 17, 46, 89, 90, 92`

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

insert:

12, 15, 17, 46, 89, 90, 92

↓ collision! where can we put 92?

| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
|-------|----|---|----|---|---|----|----|----|---|----|
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

- Remember: *it is NOT required that two non-equal objects have different hash values*
- Because of this, it is possible for two different objects to return the same hash values
  - This is called a **collision**

insert:

12, 15, 17, 46, 89, 90, 92

↓  
collision! where can we put 92?

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

# Probing

- Looking for another index to use if the preferred index is taken.
- Resolving hash collision by placing elements at other indexes in the table rather than their preferred indexes.
- How?
  - Remember the original goal

# Probing

- Looking for another index to use if the preferred index is taken.
- Resolving hash collision by placing elements at other indexes in the table rather than their preferred indexes.
- How?
  - Remember the original goal

Searching for elements is supposed to be  $O(c)$

# Probing techniques

- Linear probing
- Quadratic probing
- Separate chaining
- And more...

# Insert with linear probing

Collisions are resolved on inserts by sequentially scanning the table (with wraparound) until an empty cell is found

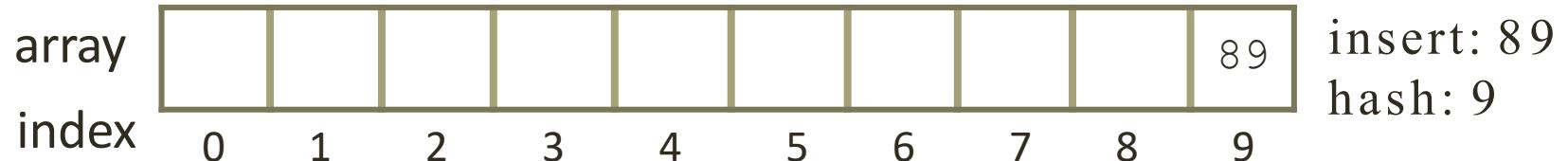
# Linear probing

- When *inserting*
  - If the spot is already taken, simply step forward one index at a time until an empty space is found
  - and, then insert item in empty space
- When *accessing*
  - Start at the hashed value index, and if this is not the item we are searching for, begin stepping forward until the item is found
  - What if we hit the end of the array?
  - When do we stop?

# Linear probing

- When *accessing*
  - Start at the hashed value index, and if this is not the item we are searching for, begin stepping forward until the item is found
  - What if we hit the end of the array? Keep looping
  - When do we stop? At first empty element

# Insert with linear probing



# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

# Insert with linear probing

|       |                                                                                     |            |         |
|-------|-------------------------------------------------------------------------------------|------------|---------|
| array |   | insert: 89 | hash: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |   | insert: 18 | hash: 8 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |   | insert: 49 | hash: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |
| array |  | insert: 58 | hash: 8 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 |            |         |

# Insert with linear probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                     |            |
|-------|-------------------------------------------------------------------------------------|------------|
| array |  | insert: 58 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 | hash: 8    |

|       |                                                                                      |           |
|-------|--------------------------------------------------------------------------------------|-----------|
| array |  | insert: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                  | hash: 9   |

# Access with linear probing

- If the table is not full, the item we seek, or an empty cell, will eventually be found
- Cost?
  - Recall that we are hoping for  $O(c)$
- Access operation follows the same path as insert...
  - if empty cell Reached, item not found
- How do we find 58?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 58 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 8    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Why?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Because items serve as placeholders during collision resolution

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table.
  - Because items serve as placeholders during collision resolution

| array | 49 | 58 | 9 |   |   |   |   | 18 |   | search: 89<br>hash: 9 |
|-------|----|----|---|---|---|---|---|----|---|-----------------------|
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8 | 9                     |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

How do we find 9?

| array | 49 | 58 | 9 |   |   |   |   |   | 18 | search: 89 |
|-------|----|----|---|---|---|---|---|---|----|------------|
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8  | hash: 9    |

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

How do we find 9?

|       |    |    |   |   |   |   |   |    |    |            |
|-------|----|----|---|---|---|---|---|----|----|------------|
| array | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |

- We must use **lazy deletion**, which marks items as deleted rather than actually removing them

# Delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

| How do we find 9? |    |    |   |   |   |   |   |    |    |            |
|-------------------|----|----|---|---|---|---|---|----|----|------------|
| array             | 49 | 58 | 9 |   |   |   |   | 18 | 89 | search: 89 |
| index             | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7  | 8  | hash: 9    |
| deleted           | F  | F  | F |   |   |   |   |    | F  | T          |

- We must use **lazy deletion**, which marks items as deleted rather than actually removing them

# Performance

If no collisions occur, then the performance of insert, delete, and access is \_\_\_\_\_

# Performance

If no collisions occur, then the performance of insert, delete, and access is **O (c)**

## Another example

|       |                                                                                                                                                                                                                                                                    |    |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|-----------------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25<br>hash: 5 |
|       | 1                                                                                                                                                                                                                                                                  | 91 | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
| index | 0                                                                                                                                                                                                                                                                  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |                       |

# Another example

|       |                                                                                                                                                                                                                                                                    |    |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|-----------------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25<br>hash: 5 |
|       | 1                                                                                                                                                                                                                                                                  | 91 | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |                       |
| index | 0                                                                                                                                                                                                                                                                  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |                       |

|       |                                                                                                                                                                                                                                                                      |    |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|---|----|---|----|---|-------|---|---|---|---|---|---|---|---|---|---|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td>25</td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |    | 1  | 91 | 71 |    | 5 | 45 | 7 | 25 | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|       | 1                                                                                                                                                                                                                                                                    | 91 | 71 |    | 5  | 45 | 7 | 25 | 9 |    |   |       |   |   |   |   |   |   |   |   |   |   |
| index | 0                                                                                                                                                                                                                                                                    | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9  |   |       |   |   |   |   |   |   |   |   |   |   |

# Another example

|       |                                                                                                                                                                                                                                                                    |         |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |            |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|----|----|----|----|---|----|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td></td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |         | 1  | 91 | 71 |    | 5 | 45 | 7 |   | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insert: 25 |
|       | 1                                                                                                                                                                                                                                                                  | 91      | 71 |    | 5  | 45 | 7 |    | 9 |   |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index | 0                                                                                                                                                                                                                                                                  | 1       | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9 |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index |                                                                                                                                                                                                                                                                    | hash: 5 |    |    |    |    |   |    |   |   |   |       |   |   |   |   |   |   |   |   |   |   |            |

|       |                                                                                                                                                                                                                                                                      |         |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |            |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|----|----|----|----|---|----|---|----|---|-------|---|---|---|---|---|---|---|---|---|---|------------|
| array | <table border="1"><tr><td></td><td>1</td><td>91</td><td>71</td><td></td><td>5</td><td>45</td><td>7</td><td>25</td><td>9</td></tr><tr><td>index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table> |         | 1  | 91 | 71 |    | 5 | 45 | 7 | 25 | 9 | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | search: 95 |
|       | 1                                                                                                                                                                                                                                                                    | 91      | 71 |    | 5  | 45 | 7 | 25 | 9 |    |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index | 0                                                                                                                                                                                                                                                                    | 1       | 2  | 3  | 4  | 5  | 6 | 7  | 8 | 9  |   |       |   |   |   |   |   |   |   |   |   |   |            |
| index |                                                                                                                                                                                                                                                                      | hash: 5 |    |    |    |    |   |    |   |    |   |       |   |   |   |   |   |   |   |   |   |   |            |

How many indices will be checked?

# Clustering

- If an item's natural spot is taken, it goes in the next open spot, making a cluster for that hash
  - **Clustering** happens because once there is a collision, there is a high probability more will occur
  - Thus, any item that hashes into a cluster will require several attempts to resolve the collision

| array | 0 | 1 | 2  | 3  | 4 | 5 | 6  | 7 | 8  | 9 |
|-------|---|---|----|----|---|---|----|---|----|---|
| index |   | 1 | 91 | 71 |   | 5 | 45 | 7 | 25 | 9 |

Let  $h(k)$  be a [hash function](#) that maps an element  $k$  to an integer in  $[0, m-1]$ , where  $m$  is the size of the table. Let the  $i^{\text{th}}$  probe position for a value  $k$  be given by the function

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \pmod{m}$$

where  $c_2 \neq 0$ . If  $c_2 = 0$ , then  $h(k, i)$  degrades to a [linear probe](#). For a given [hash table](#), the values of  $c_1$  and  $c_2$  remain constant.

# Quadratic probing

- Probing is how to resolve collisions
  - to address clustering, we will use **quadratic probing**
- If  $\text{hash}(\text{key}) = H$ , and the cell at index  $H$  is occupied:
  - Try  $(H+1^2) \bmod \text{size}$
  - Then  $(H+2^2) \bmod \text{size}$
  - Then  $(H+3^2) \bmod \text{size}$
  - and so on...
  - Wrap around to beginning of array if necessary

# Quadratic Probing...

The screenshot shows a web browser window titled "Hashing Quadratic Probing Ani". The URL is [www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html](http://www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html). The page content is titled "Hashing Using Quadratic Probing Animation by Y. Daniel Liang". A usage instruction is provided: "Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold factor and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99." Below this, a table shows the current state of a hash table of size 11. The table has 11 slots, indexed [0] to [10]. Slots [0], [4], [5], [6], [7], [8], [9], and [10] are empty. Slots [1], [2], [3], and [11] contain the values 44, 4, 16, and 28 respectively. At the bottom, there are input fields for "Enter Initial Table Size" (set to 11) and "Enter a Load Factor Threshold" (set to 0.75). Below these are buttons for "Enter a key:", "Search", "Insert", "Remove", and "Remove All".

Source:

<http://www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html>

(<http://www.cs.armstrong.edu/liang/animation/web/LinearProbing.html>)

# Concerns...

- Is quadratic probing guaranteed to find an open spot?  
can it search the same spot twice?
- Suppose the array size is 16, and hash (key) = 0

$$0 \% 16 = 0$$

$$(0+1^2) \% 16 = 1$$

$$(0+2^2) \% 16 = 4$$

$$(0+3^2) \% 16 = 9$$

$$(0+4^2) \% 16 = 0$$

$$(0+5^2) \% 16 = 9$$

$$(0+6^2) \% 16 = 4$$

$$(0+7^2) \% 16 = 1$$

## Concerns...

- Is quadratic probing guaranteed to find an open spot?  
can it search the same spot twice?
- Suppose the array size is 16, and hash (key) = 0

$$0 \% 16 = 0$$

$$(0+1^2) \% 16 = 1$$

$$(0+2^2) \% 16 = 4$$

$$(0+3^2) \% 16 = 9$$

$$(0+4^2) \% 16 = 0$$

$$(0+5^2) \% 16 = 9$$

$$(0+6^2) \% 16 = 4$$

$$(0+7^2) \% 16 = 1$$

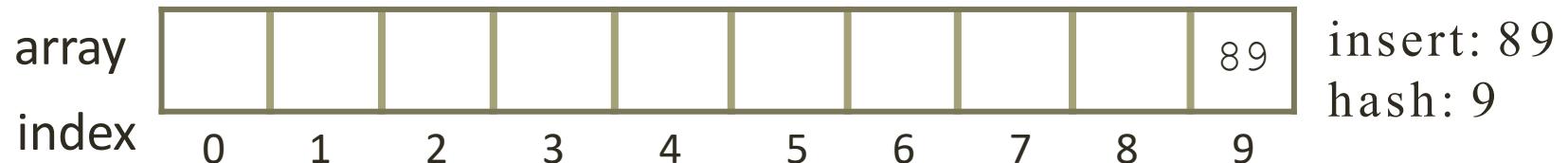
**limitation:** at most, half of the table can be used to resolve collisions

once table is half full it is difficult to find an empty spot

...Called **secondary clustering**

# Insert with quadratic probing

# Insert with quadratic probing

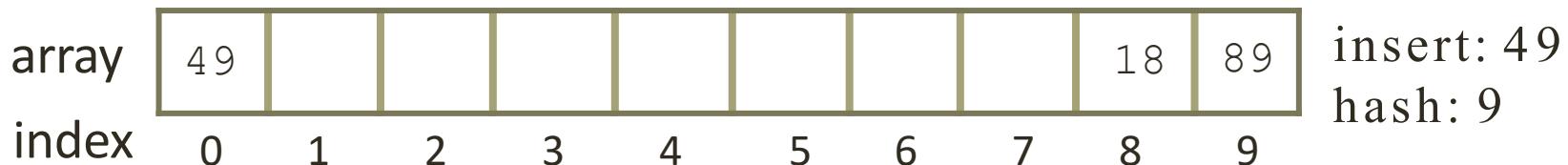
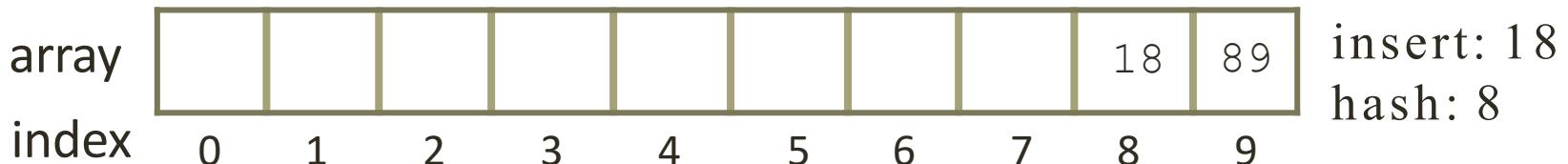
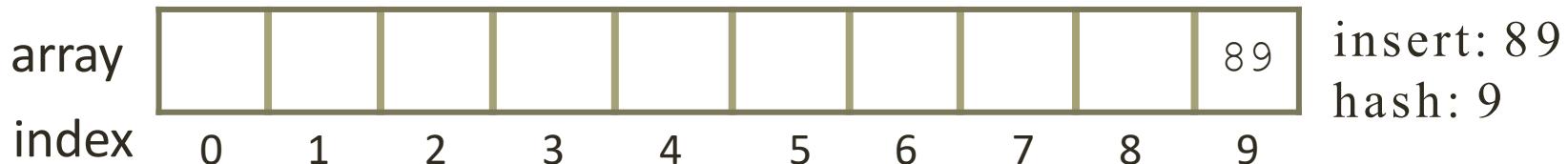


# Insert with quadratic probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

# Insert with quadratic probing



# Insert with quadratic probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                     |            |
|-------|-------------------------------------------------------------------------------------|------------|
| array |  | insert: 58 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 | hash: 8    |

# Insert with quadratic probing

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 89 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 18 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 8    |

|       |                                                                                    |            |
|-------|------------------------------------------------------------------------------------|------------|
| array |  | insert: 49 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                | hash: 9    |

|       |                                                                                     |            |
|-------|-------------------------------------------------------------------------------------|------------|
| array |  | insert: 58 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                 | hash: 8    |

|       |                                                                                      |           |
|-------|--------------------------------------------------------------------------------------|-----------|
| array |  | insert: 9 |
| index | 0 1 2 3 4 5 6 7 8 9                                                                  | hash: 9   |

|       |                                                                                                                                                          |    |    |   |    |    |    |    |    |    |    |    |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|---|----|----|----|----|----|----|----|----|
| array | <table border="1"> <tr> <td>77</td><td>11</td><td></td><td>33</td><td>4</td><td>16</td><td>28</td><td></td><td></td><td>22</td><td>21</td></tr> </table> | 77 | 11 |   | 33 | 4  | 16 | 28 |    |    | 22 | 21 |
| 77    | 11                                                                                                                                                       |    | 33 | 4 | 16 | 28 |    |    | 22 | 21 |    |    |
| index | 0    1    2    3    4    5    6    7    8    9    10                                                                                                     |    |    |   |    |    |    |    |    |    |    |    |

- Try  $(H+1^2) \bmod \text{size}$
  - Then  $(H+2^2) \bmod \text{size}$
  - Then  $(H+3^2) \bmod \text{size}$
- insert: 55  
hash: 0

Can quadratic probing find a spot?!

...solution!

Rehashing

# ...solution!

- Resize the table (array)
  - Just like resizing an array, we resize the table to be larger
    - Usually a larger prime number
  - Instead of a simple copy-everything-over, all items must be rehashed
    - why?

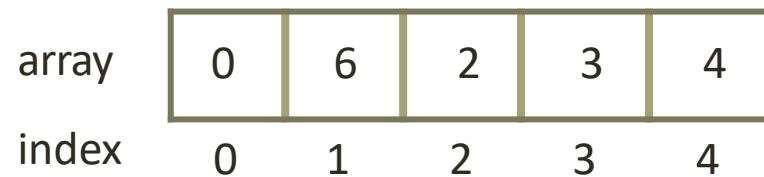
# ...solution!

- Resize the table (array)
  - Just like resizing an array, we resize the table to be larger
    - Usually a larger prime number
  - Instead of a simple copy-everything-over, all items must be rehashed
    - why?

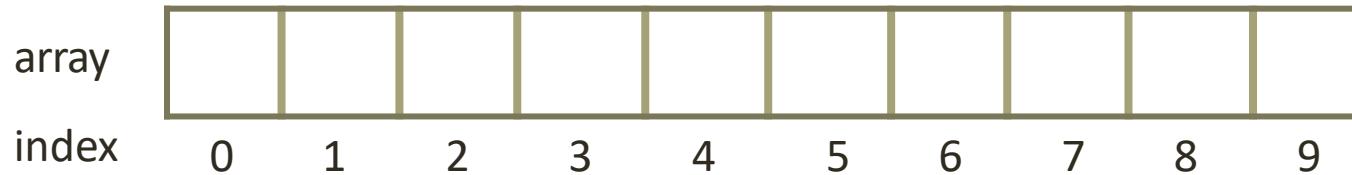
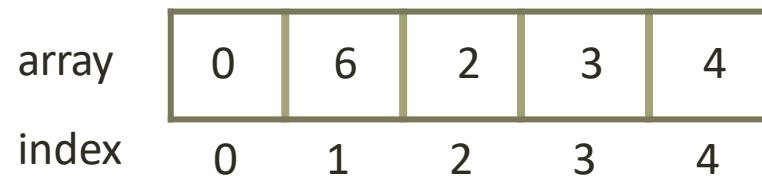
# Rehashing

- Resizing a hash table to increase its capacity and enabling it to store more elements, or store them more efficiently

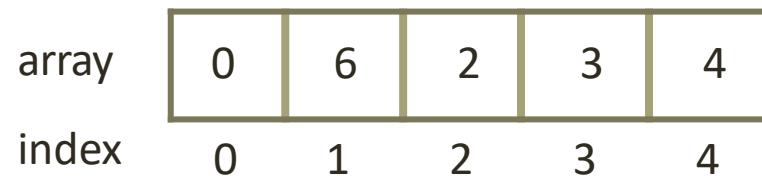
# Rehashing example



# Rehashing example



# Rehashing example



# Load factor

- The measure of how full the hash table is
- Table size relative to the array capacity
  - the fraction of the table that is full
    - Number of items in array / size of array
    - Called  $\lambda$
    - $0 \leq \lambda \leq 1$
- Instead of rehashing when the table is half full, rehash when  $\lambda$  becomes large
- Standard for Java Hashmap is .75

# Rehashing

- Resizing a hash table to increase its capacity and enabling it to store more elements, or store them more efficiently

Can you think of an alternative for collision management?

# Separate chaining

- Why not make each spot in the array capable of holding more than one item?

# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array

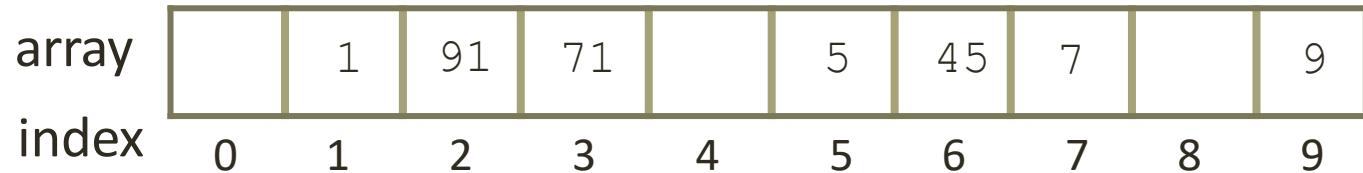
# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array
- For insertion, append the item to the end of the list
  - Insertion is  $O(c)$  if we have what?

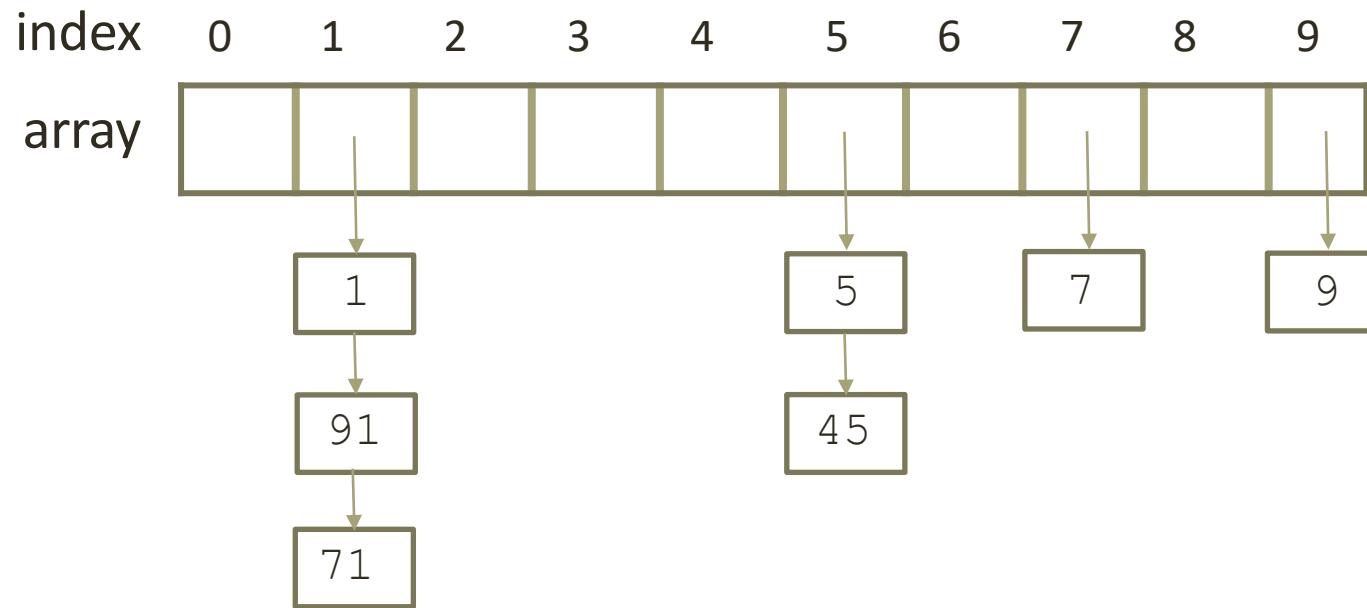
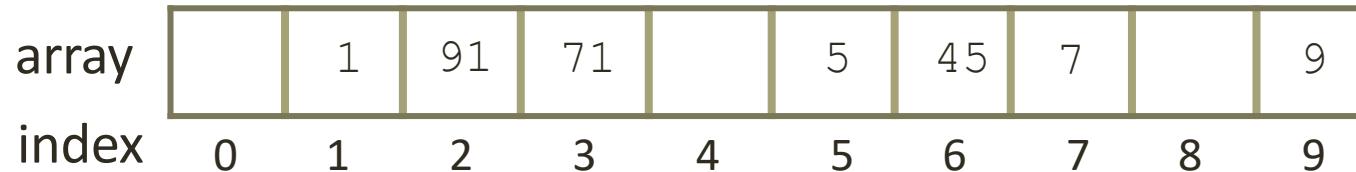
# Separate chaining

- Why not make each spot in the array capable of holding more than one item?
  - Use an array of linked lists
  - Hash function selects index into array
- For insertion, append the item to the end of the list
  - Insertion is  $O(c)$  if we have what?
- Accessing is a linear scan through the list
  - Fast if the list is short

# Separate chaining



# Separate chaining



What data structure we can use for implementation?

# Performance

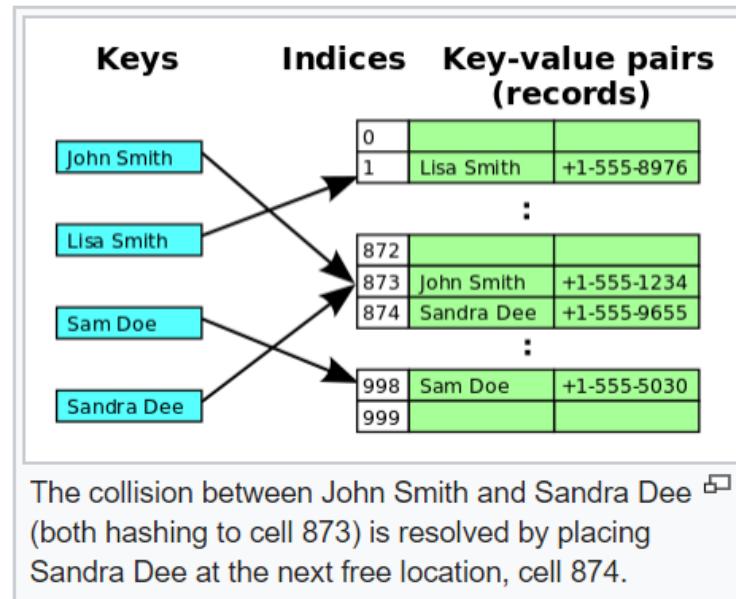
- Clusters can still happen with separate chaining
- When to resize?
  - One easy rule
    - resize the hash table once its size exceed its capacity.
  - However, most hash table implementation resize before that
    - How?

# Load factor

- The measure of how full the hash table is
- Table size relative to the array capacity
  - the fraction of the table that is full
  - Called  $\lambda$
  - $0 \leq \lambda \leq 1$
- Instead of rehashing when the table is half full, rehash when  $\lambda$  becomes large

# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?



# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?
- One solution is to somehow generate an integer from a string
  - Length of string?
  - Sum of all characters?
  - Some combination of both?

# What about data without natural indices?

- How can we do this for non-integer items?
  - Integers have an obvious solution...
  - Use the integer itself as the index
  - What index should use for, say, a char?
- One solution is to somehow generate an integer from a string
  - Length of string?
  - Sum of all characters?
  - Some combination of both?
- A method that generates an integer index given any object is called a ***hash function***

A couple of rules...

- Always returns the same number for the same object

## A couple of rules...

- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects

# A couple of rules...

- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects
- Good hash functions return evenly distributed numbers for the input items

## A couple of rules...

- Always returns the same number for the same object
- If `object1.equals(object2) == true`
  - MUST return the same integer for both objects
- Good hash functions return evenly distributed numbers for the input items
- *It is not required that two non-equal objects have different hash values*

# Java's hashCode

- Every Object in Java has a method hashCode
- Returns an integer based on the object
- Default for this method (if you don't override it) is to return the memory address of the object
- Will not be very well-distributed if your items are contiguous in memory

A bit more on hash functions...

- ints have an obvious hash value

|       |    |   |    |   |   |    |    |    |   |    |
|-------|----|---|----|---|---|----|----|----|---|----|
| array | 90 |   | 12 |   |   | 15 | 46 | 17 |   | 89 |
| index | 0  | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8 | 9  |

- What about Strings? Books? Shapes?...
- We must not overlook the requirement of a *good* hash functions

# Remember...

- Hash functions take any item as input and produce an integer as output
- Given the same input the function always returns the same output
- Two different inputs MAY have the same hash value

# Thinking about **chars** and **Strings**

- ASCII defines an encoding for characters
  - 'a' = 97
  - 'b' = 98
  - ...
  - 'z' = 122
  - '2' = 50 - ...
- The `char` type is actually just a small integer
  - 8 bits (if using ASCII, 16 bits if using UTF-16 coding for the `char`) instead of the usual 32

| Dec | Hx | Oct | Char       |                          | Dec | Hx | Oct | Html  | Chr          |  | Dec | Hx | Oct | Html  | Chr        |
|-----|----|-----|------------|--------------------------|-----|----|-----|-------|--------------|--|-----|----|-----|-------|------------|
| 0   | 0  | 000 | <b>NUL</b> | (null)                   | 32  | 20 | 040 | &#32; | <b>Space</b> |  | 64  | 40 | 100 | &#64; | <b>Ø</b>   |
| 1   | 1  | 001 | <b>SOH</b> | (start of heading)       | 33  | 21 | 041 | &#33; | <b>!</b>     |  | 65  | 41 | 101 | &#65; | <b>A</b>   |
| 2   | 2  | 002 | <b>STX</b> | (start of text)          | 34  | 22 | 042 | &#34; | <b>"</b>     |  | 66  | 42 | 102 | &#66; | <b>B</b>   |
| 3   | 3  | 003 | <b>ETX</b> | (end of text)            | 35  | 23 | 043 | &#35; | <b>#</b>     |  | 67  | 43 | 103 | &#67; | <b>C</b>   |
| 4   | 4  | 004 | <b>EOT</b> | (end of transmission)    | 36  | 24 | 044 | &#36; | <b>\$</b>    |  | 68  | 44 | 104 | &#68; | <b>D</b>   |
| 5   | 5  | 005 | <b>ENQ</b> | (enquiry)                | 37  | 25 | 045 | &#37; | <b>%</b>     |  | 69  | 45 | 105 | &#69; | <b>E</b>   |
| 6   | 6  | 006 | <b>ACK</b> | (acknowledge)            | 38  | 26 | 046 | &#38; | <b>&amp;</b> |  | 70  | 46 | 106 | &#70; | <b>F</b>   |
| 7   | 7  | 007 | <b>BEL</b> | (bell)                   | 39  | 27 | 047 | &#39; | <b>'</b>     |  | 71  | 47 | 107 | &#71; | <b>G</b>   |
| 8   | 8  | 010 | <b>BS</b>  | (backspace)              | 40  | 28 | 050 | &#40; | <b>(</b>     |  | 72  | 48 | 110 | &#72; | <b>H</b>   |
| 9   | 9  | 011 | <b>TAB</b> | (horizontal tab)         | 41  | 29 | 051 | &#41; | <b>)</b>     |  | 73  | 49 | 111 | &#73; | <b>I</b>   |
| 10  | A  | 012 | <b>LF</b>  | (NL line feed, new line) | 42  | 2A | 052 | &#42; | <b>*</b>     |  | 74  | 4A | 112 | &#74; | <b>J</b>   |
| 11  | B  | 013 | <b>VT</b>  | (vertical tab)           | 43  | 2B | 053 | &#43; | <b>+</b>     |  | 75  | 4B | 113 | &#75; | <b>K</b>   |
| 12  | C  | 014 | <b>FF</b>  | (NP form feed, new page) | 44  | 2C | 054 | &#44; | <b>,</b>     |  | 76  | 4C | 114 | &#76; | <b>L</b>   |
| 13  | D  | 015 | <b>CR</b>  | (carriage return)        | 45  | 2D | 055 | &#45; | <b>-</b>     |  | 77  | 4D | 115 | &#77; | <b>M</b>   |
| 14  | E  | 016 | <b>SO</b>  | (shift out)              | 46  | 2E | 056 | &#46; | <b>.</b>     |  | 78  | 4E | 116 | &#78; | <b>N</b>   |
| 15  | F  | 017 | <b>SI</b>  | (shift in)               | 47  | 2F | 057 | &#47; | <b>/</b>     |  | 79  | 4F | 117 | &#79; | <b>O</b>   |
| 16  | 10 | 020 | <b>DLE</b> | (data link escape)       | 48  | 30 | 060 | &#48; | <b>Ø</b>     |  | 80  | 50 | 120 | &#80; | <b>P</b>   |
| 17  | 11 | 021 | <b>DC1</b> | (device control 1)       | 49  | 31 | 061 | &#49; | <b>1</b>     |  | 81  | 51 | 121 | &#81; | <b>Q</b>   |
| 18  | 12 | 022 | <b>DC2</b> | (device control 2)       | 50  | 32 | 062 | &#50; | <b>2</b>     |  | 82  | 52 | 122 | &#82; | <b>R</b>   |
| 19  | 13 | 023 | <b>DC3</b> | (device control 3)       | 51  | 33 | 063 | &#51; | <b>3</b>     |  | 83  | 53 | 123 | &#83; | <b>S</b>   |
| 20  | 14 | 024 | <b>DC4</b> | (device control 4)       | 52  | 34 | 064 | &#52; | <b>4</b>     |  | 84  | 54 | 124 | &#84; | <b>T</b>   |
| 21  | 15 | 025 | <b>NAK</b> | (negative acknowledge)   | 53  | 35 | 065 | &#53; | <b>5</b>     |  | 85  | 55 | 125 | &#85; | <b>U</b>   |
| 22  | 16 | 026 | <b>SYN</b> | (synchronous idle)       | 54  | 36 | 066 | &#54; | <b>6</b>     |  | 86  | 56 | 126 | &#86; | <b>V</b>   |
| 23  | 17 | 027 | <b>ETB</b> | (end of trans. block)    | 55  | 37 | 067 | &#55; | <b>7</b>     |  | 87  | 57 | 127 | &#87; | <b>W</b>   |
| 24  | 18 | 030 | <b>CAN</b> | (cancel)                 | 56  | 38 | 070 | &#56; | <b>8</b>     |  | 88  | 58 | 130 | &#88; | <b>X</b>   |
| 25  | 19 | 031 | <b>EM</b>  | (end of medium)          | 57  | 39 | 071 | &#57; | <b>9</b>     |  | 89  | 59 | 131 | &#89; | <b>Y</b>   |
| 26  | 1A | 032 | <b>SUB</b> | (substitute)             | 58  | 3A | 072 | &#58; | <b>:</b>     |  | 90  | 5A | 132 | &#90; | <b>Z</b>   |
| 27  | 1B | 033 | <b>ESC</b> | (escape)                 | 59  | 3B | 073 | &#59; | <b>:</b>     |  | 91  | 5B | 133 | &#91; | <b>[</b>   |
| 28  | 1C | 034 | <b>FS</b>  | (file separator)         | 60  | 3C | 074 | &#60; | <b>&lt;</b>  |  | 92  | 5C | 134 | &#92; | <b>\</b>   |
| 29  | 1D | 035 | <b>GS</b>  | (group separator)        | 61  | 3D | 075 | &#61; | <b>=</b>     |  | 93  | 5D | 135 | &#93; | <b>]</b>   |
| 30  | 1E | 036 | <b>RS</b>  | (record separator)       | 62  | 3E | 076 | &#62; | <b>&gt;</b>  |  | 94  | 5E | 136 | &#94; | <b>^</b>   |
| 31  | 1F | 037 | <b>US</b>  | (unit separator)         | 63  | 3F | 077 | &#63; | <b>?</b>     |  | 95  | 5F | 137 | &#95; | <b>_</b>   |
|     |    |     |            |                          |     |    |     |       |              |  |     |    |     |       | <b>DEL</b> |

- A String is essentially an array of char

```
String s = "hello";
```



- Java hides these details
- How can we use this to create a hash function for Strings?

- A String is essentially an array of char

```
String s = "hello";
```



- Java hides these details
- How can we use this to create a hash function for Strings?
  - How about we add all the values?

- A String is essentially an array of char

```
String s = "hello";
```

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 104 | 101 | 108 | 108 | 111 |
|-----|-----|-----|-----|-----|

- Java hides these details
- How can we use this to create a hash function for Strings?
  - How about we add all the values? → Not great!
  - Why?
  - It's actually more like this:
    - $h(s) = s[0]*31^{n-1} + s[1]*31^{n-2} + \dots + s[n-1]$

The good and widely used way to define the hash of a string  $s$  of length  $n$  is

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \dots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

where  $p$  and  $m$  are some chosen, positive numbers. It is called a **polynomial rolling hash function**.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems
- Quadratic probing
  - Has lesser clustering problems
  - Requires  $\lambda < 0.5$ , and prime table size

# Review

- $O(c)$  for all major operations
  - assuming  $\lambda$  is managed
- Linear probing
  - Has clustering problems
- Quadratic probing
  - Has lesser clustering problems
  - Requires  $\lambda < 0.5$ , and prime table size
- Separate chaining
  - Probably the easiest to implement, as well as the best performing

**What is the load factor  $\lambda$  for the following hash table?**

- A) 4
- B) 6
- C) 0.4
- D) 0.5
- E) 0.6

| array | 104 | 34 |   | 19 | 111 | 98 |   | 52 |   |   |
|-------|-----|----|---|----|-----|----|---|----|---|---|
| index | 0   | 1  | 2 | 3  | 4   | 5  | 6 | 7  | 8 | 9 |

## Using linear probing, in what index will item 93 be added?

- A) 1
- B) 5
- C) 6
- D) 7

|       |    |   |   |    |    |   |   |    |    |
|-------|----|---|---|----|----|---|---|----|----|
| array | 49 |   | 9 | 58 | 34 |   |   | 18 | 89 |
| index | 0  | 1 | 2 | 3  | 4  | 5 | 6 | 7  | 8  |

**Using linear probing, in what index will item 22 be added?**

- A) 1
- B) 5
- C) 6
- D) 7

|       |    |   |   |    |    |   |   |    |    |
|-------|----|---|---|----|----|---|---|----|----|
| array | 49 |   | 9 | 58 | 34 |   |   | 18 | 89 |
| index | 0  | 1 | 2 | 3  | 4  | 5 | 6 | 7  | 8  |

# Recap

- Hash tables
  - collection structure with  $O(c)$  for major operations
- but!...
  - Hash function must minimize collisions
    - *Should evenly distribute values across all possible integers*
  - Collisions must be carefully dealt with
  - Hash function runtime must be fast
- No ordering
  - *How do we find the smallest item in a hash table?*
  - *In a BST?*

# Graphs

## CSC220|Computer Programming 2

Last Time...

# Heap sort

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- How?

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array ☺

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array 😊 → Why?!

# Pseudo code...

A = array to be sorted

H = create new heap

For each element n in A

    add n to H

While (H not empty)

    remove element from H

    add element back into A

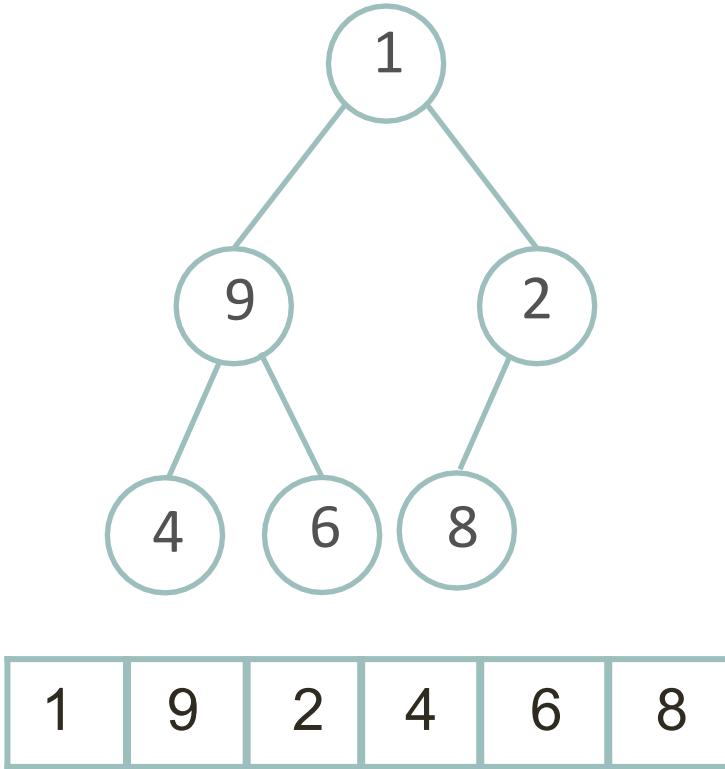
# In-place heap sort

- Use the original array itself as the heap
- The idea is to treat the array as an initially invalid heap and repair it into a proper heap
  - Bubbling elements into their proper position
- Remove elements from the heap and put them at the end of the array.
- min-heap gives you descending order and max-heap gives you ascending order

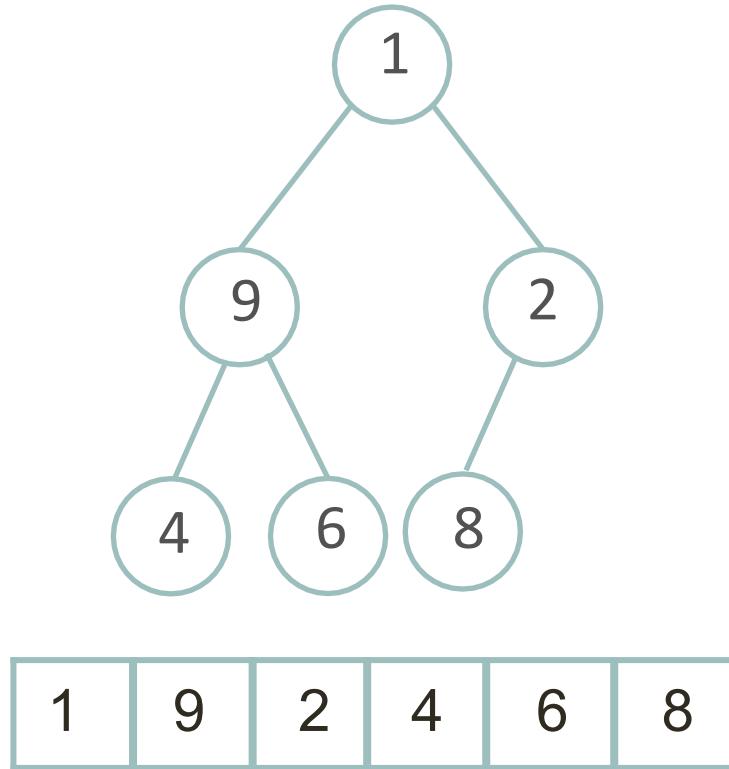
# In-place heap sort

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 9 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|

# In-place heap sort

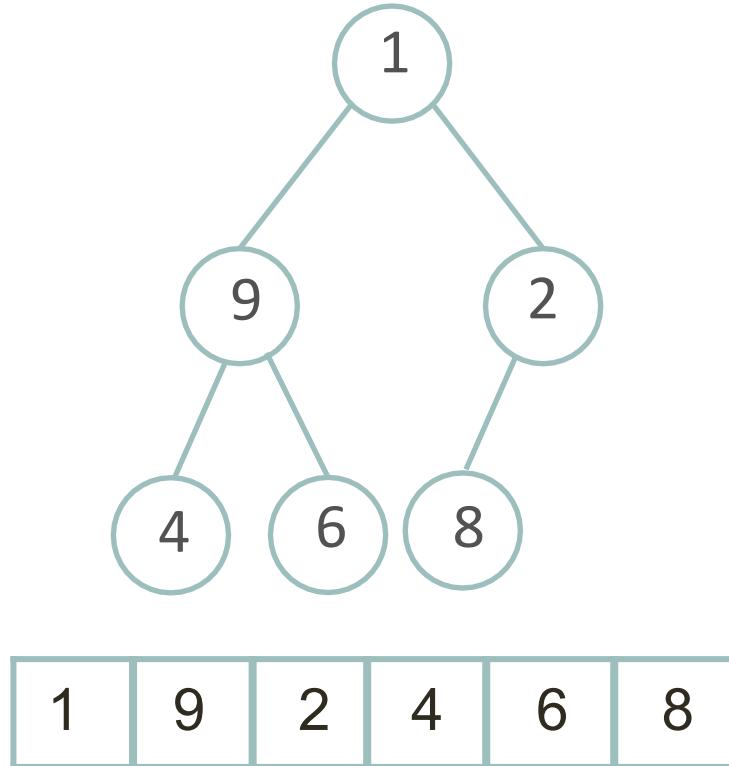


# In-place heap sort



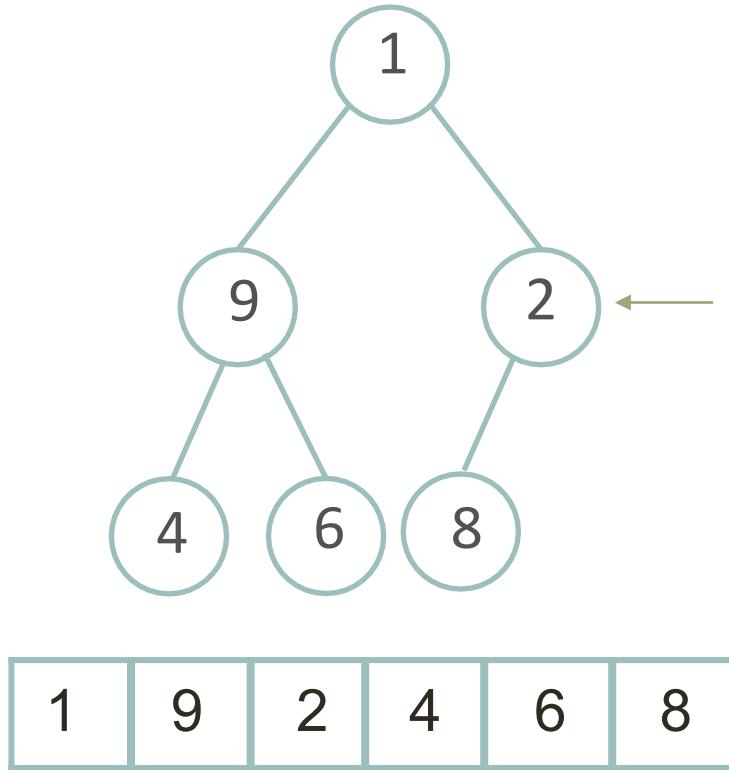
Not a heap → Need to *heapify*

# In-place heap sort



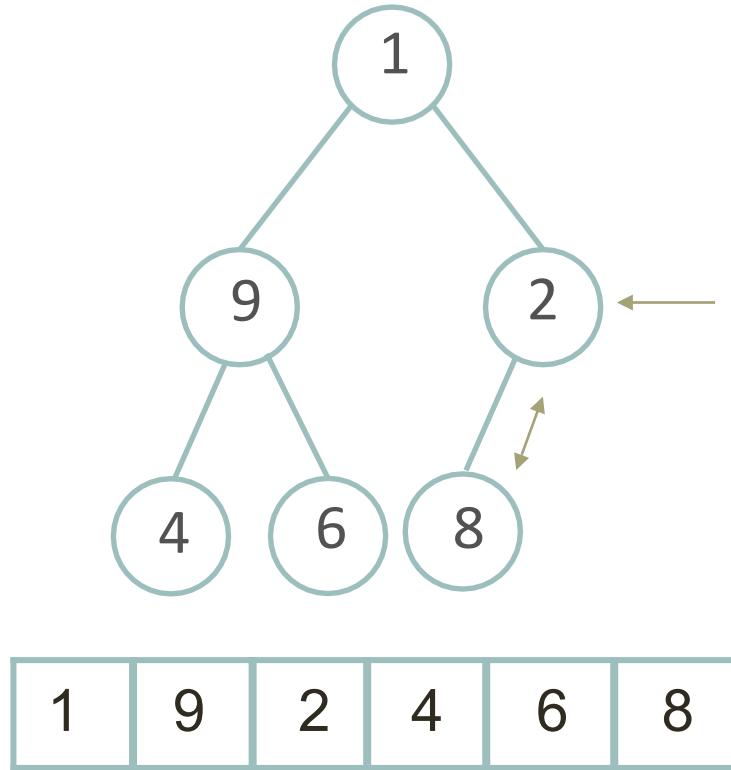
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



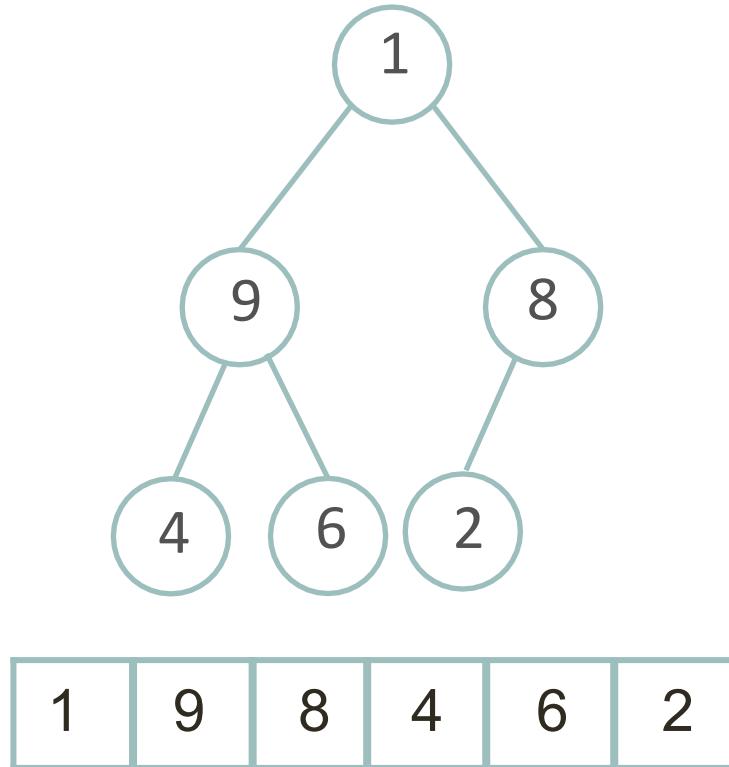
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



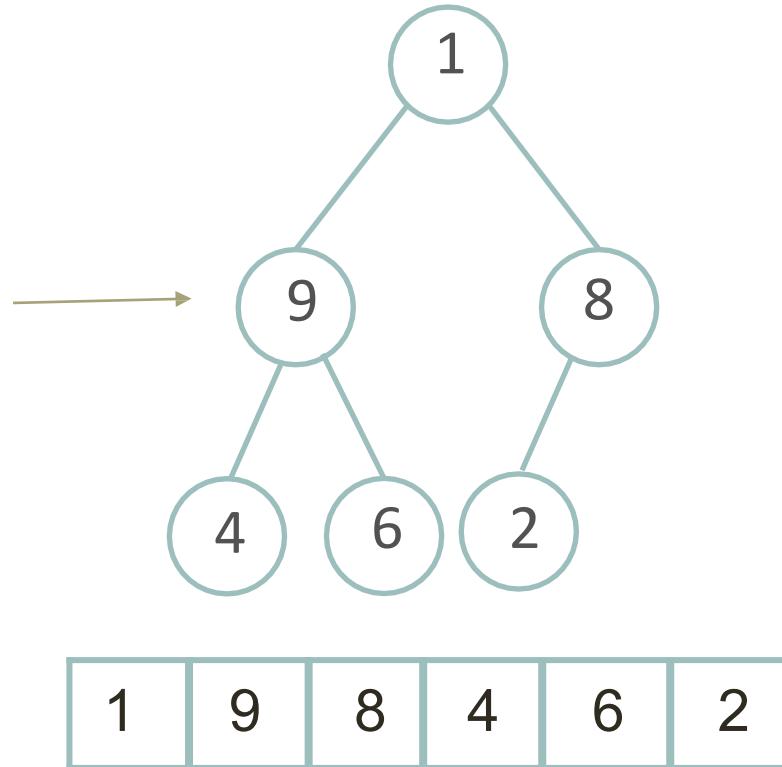
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



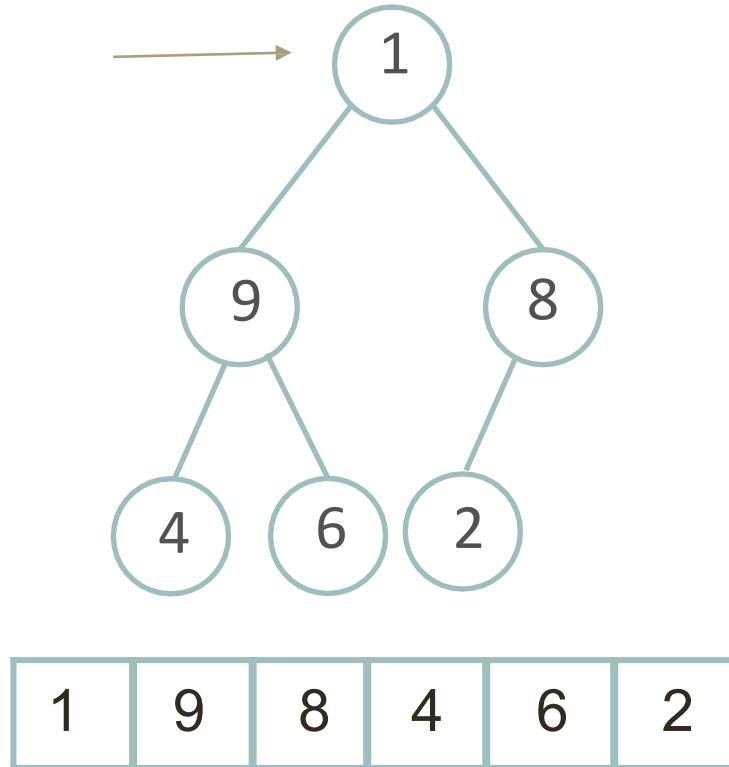
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



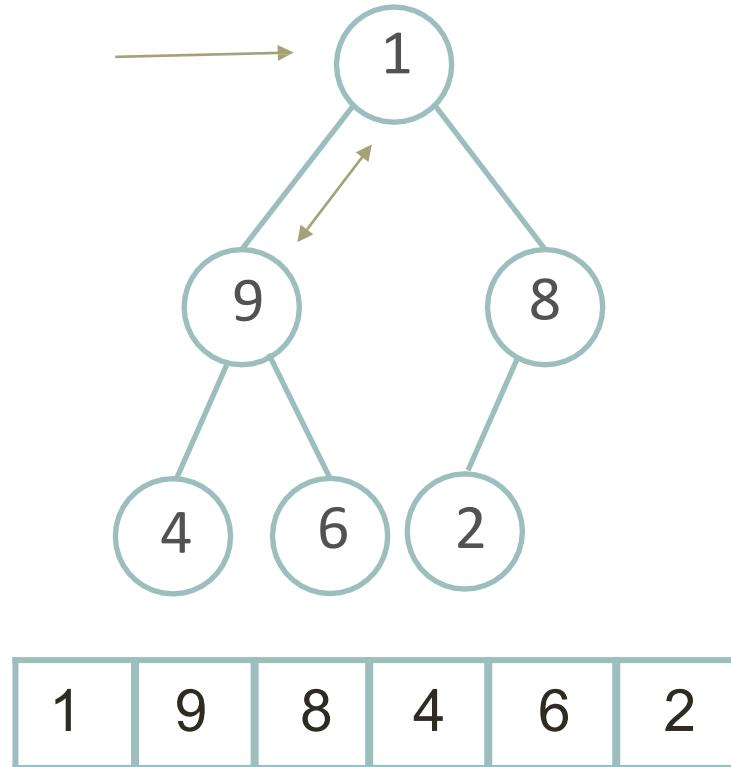
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



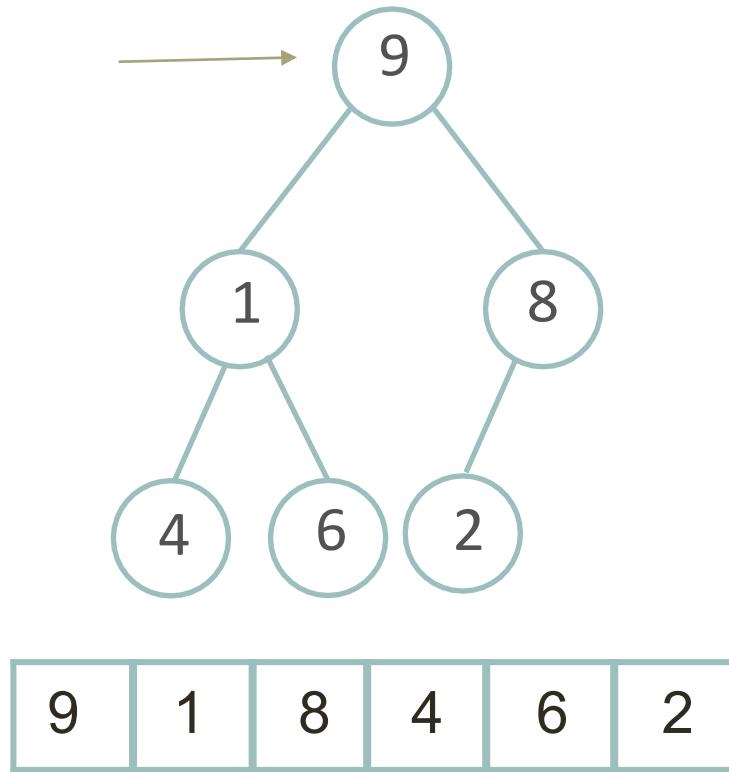
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



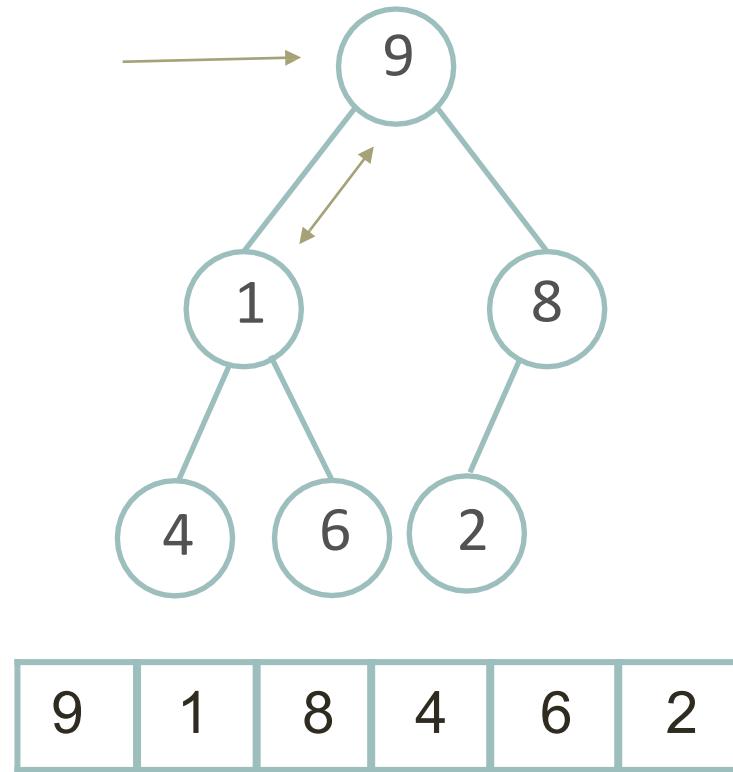
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



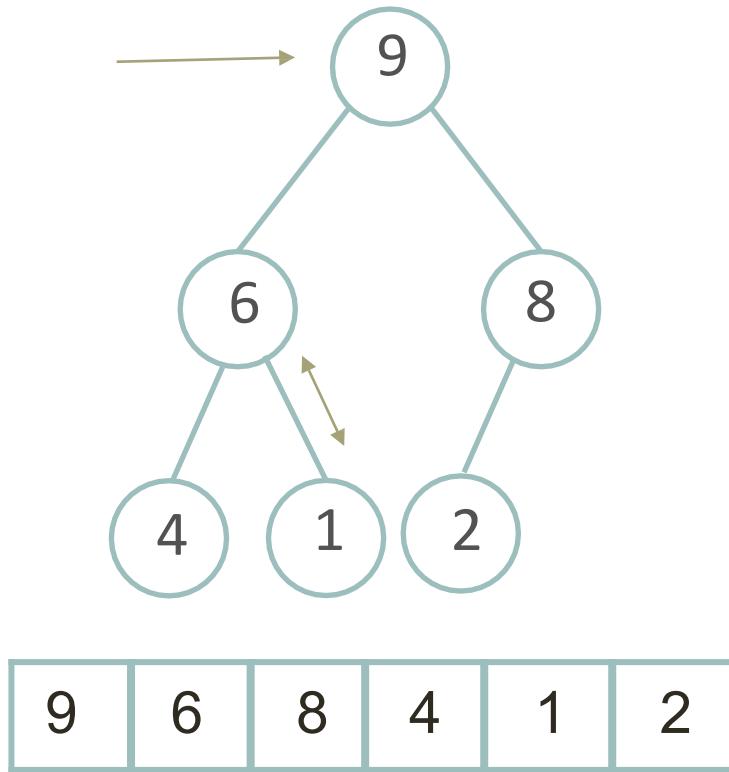
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

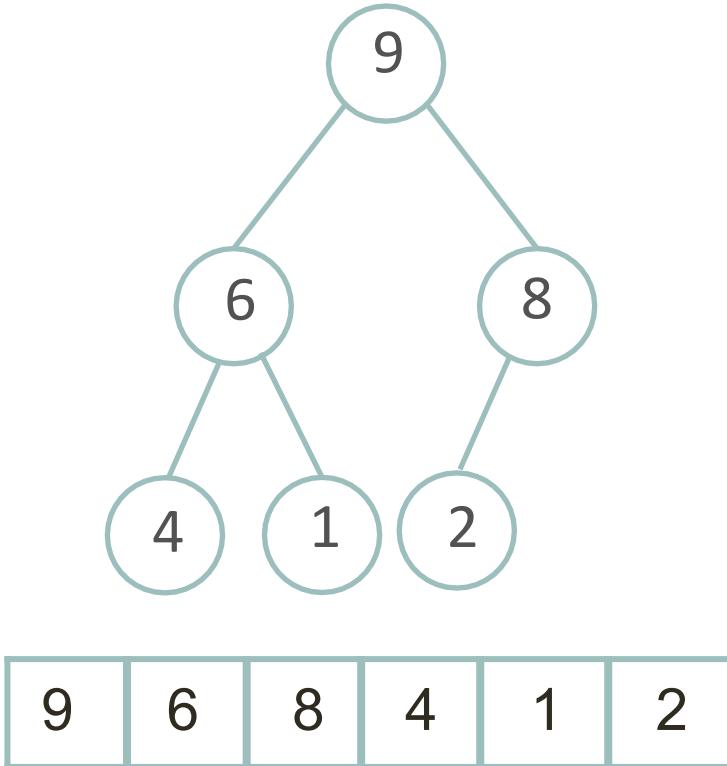


Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

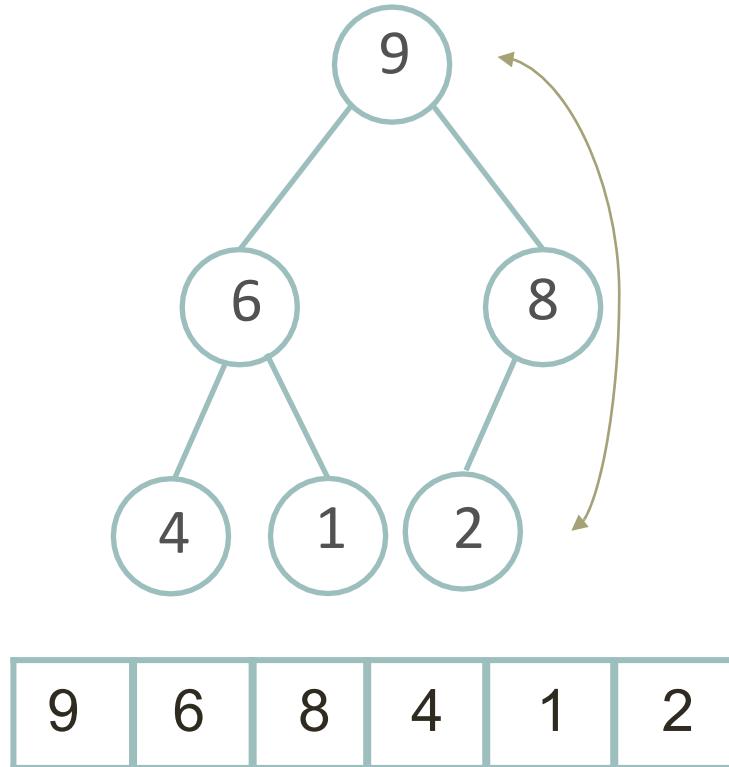


# In-place heap sort



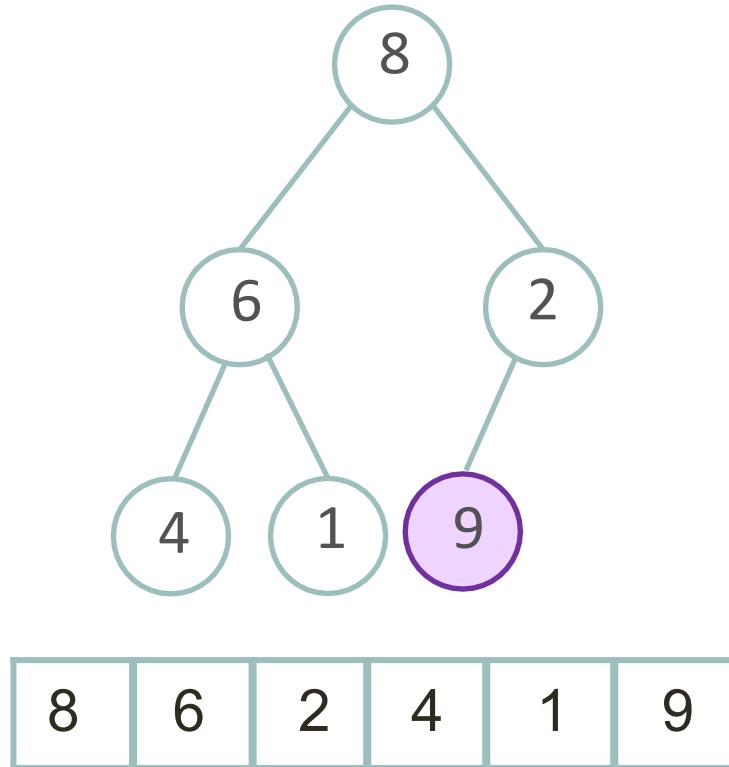
Not yet sorted array!

# In-place heap sort



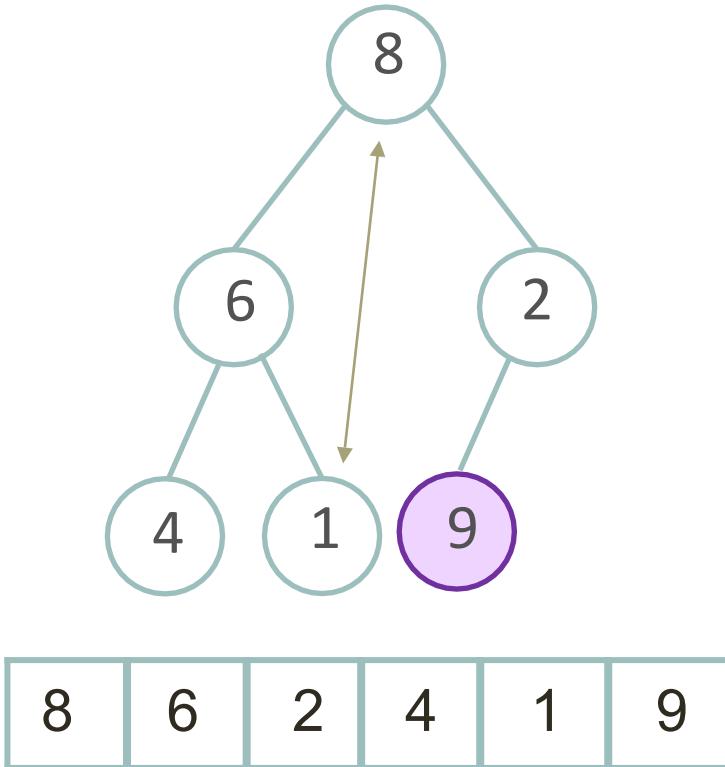
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



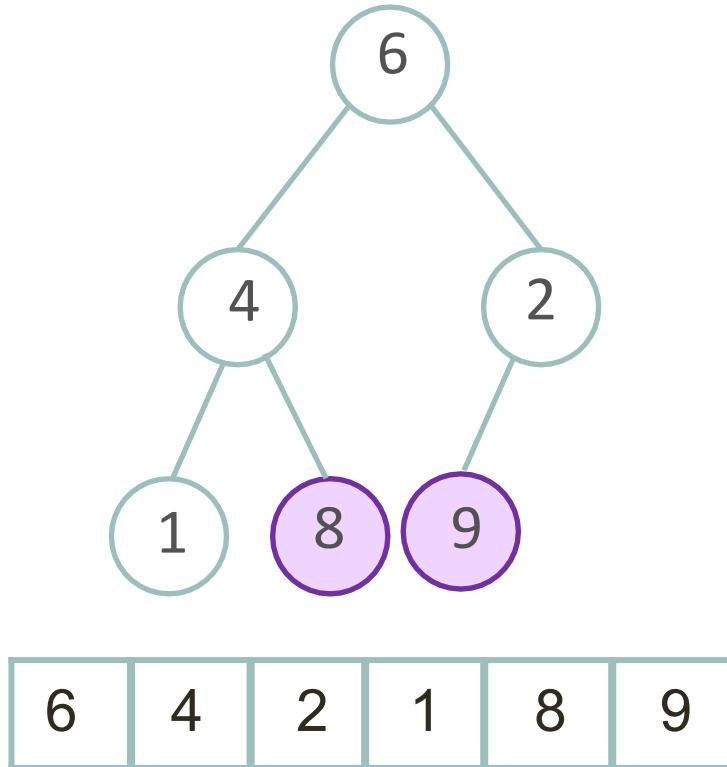
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



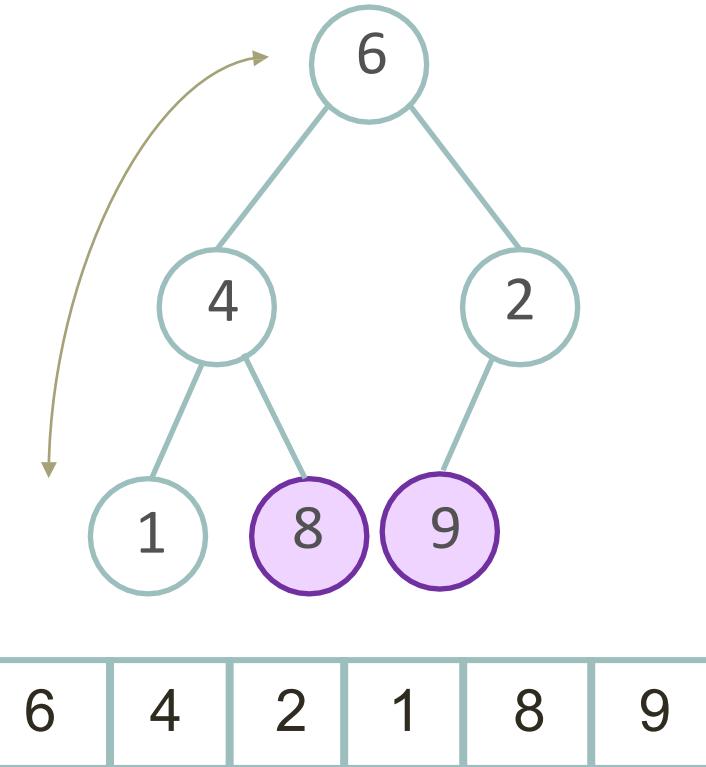
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



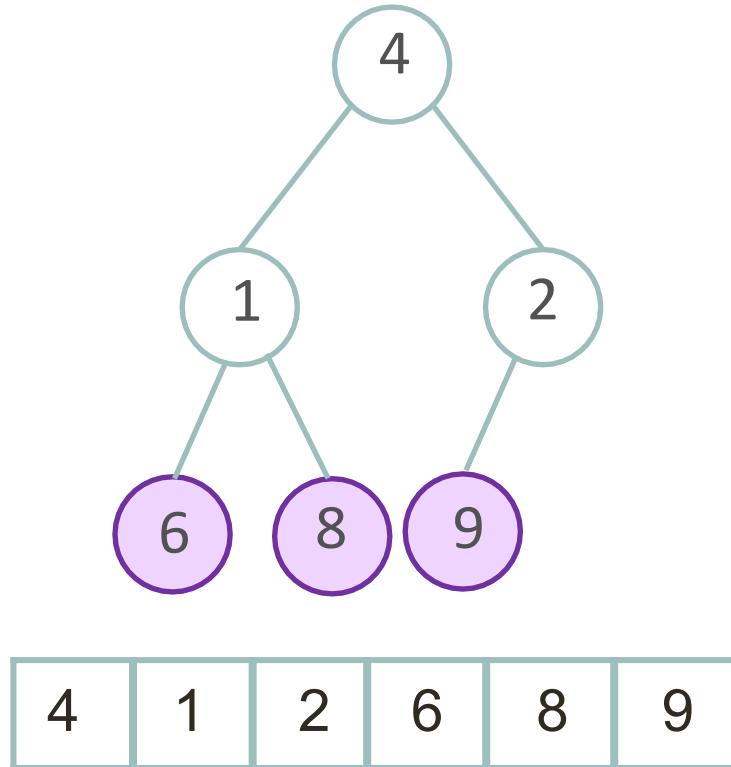
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



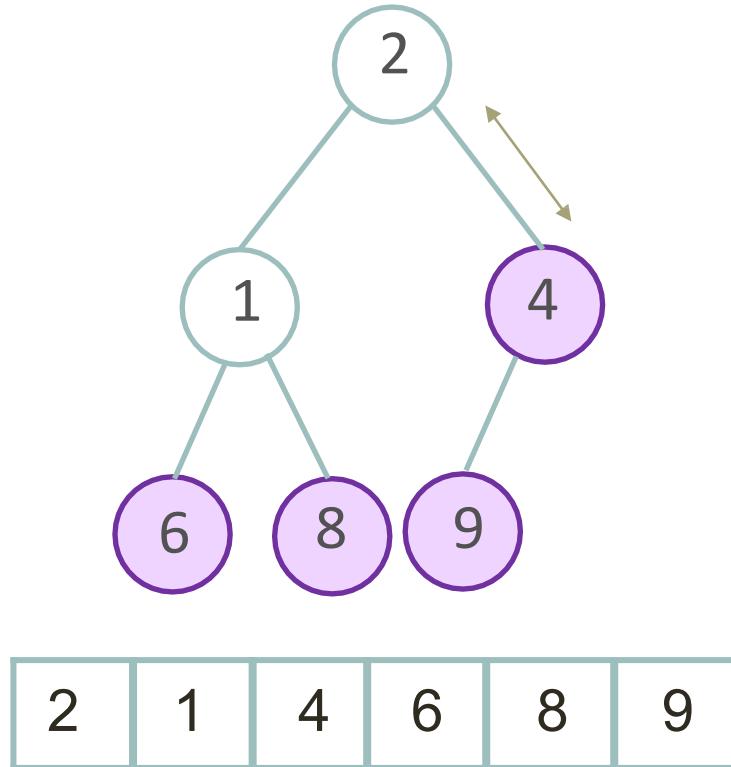
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



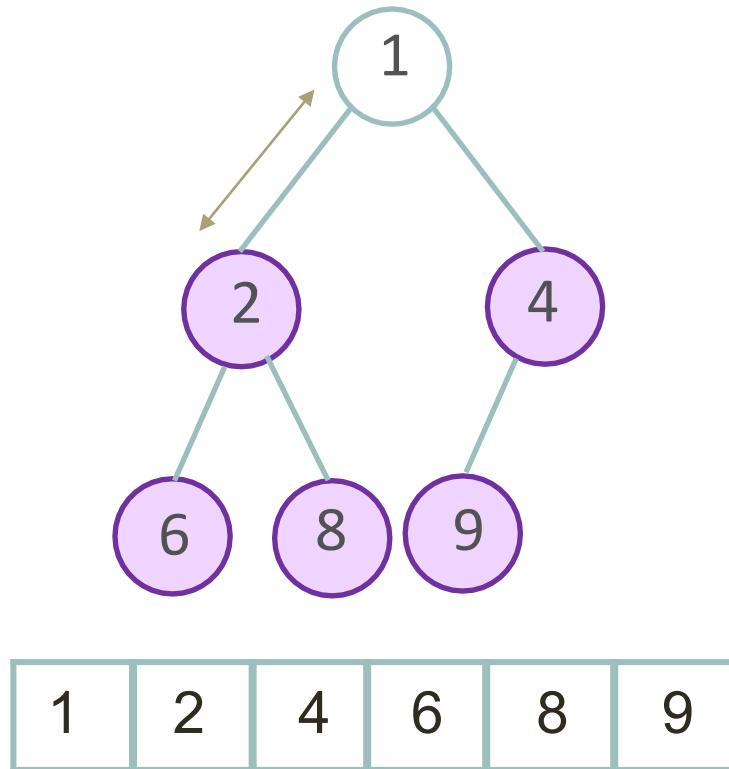
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



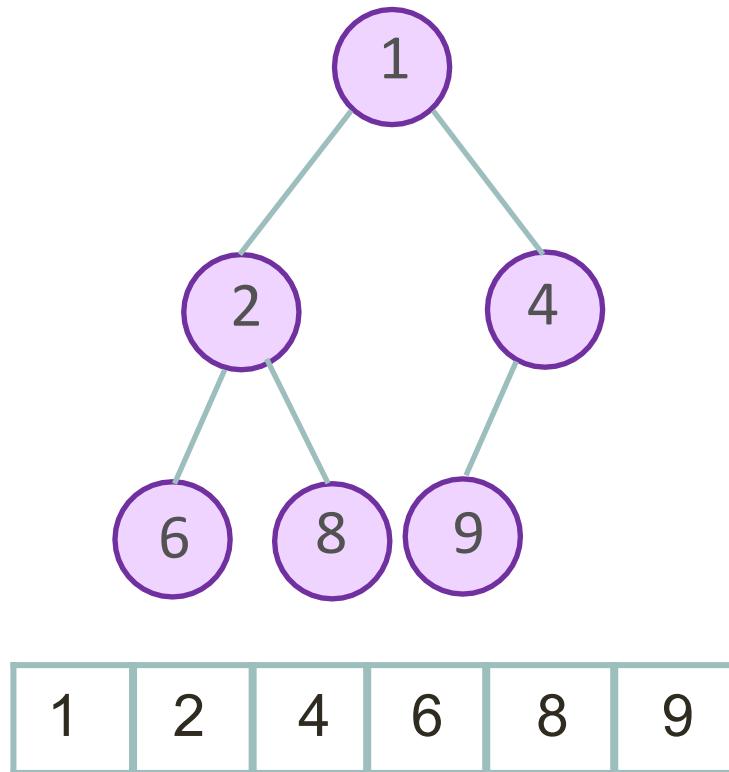
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

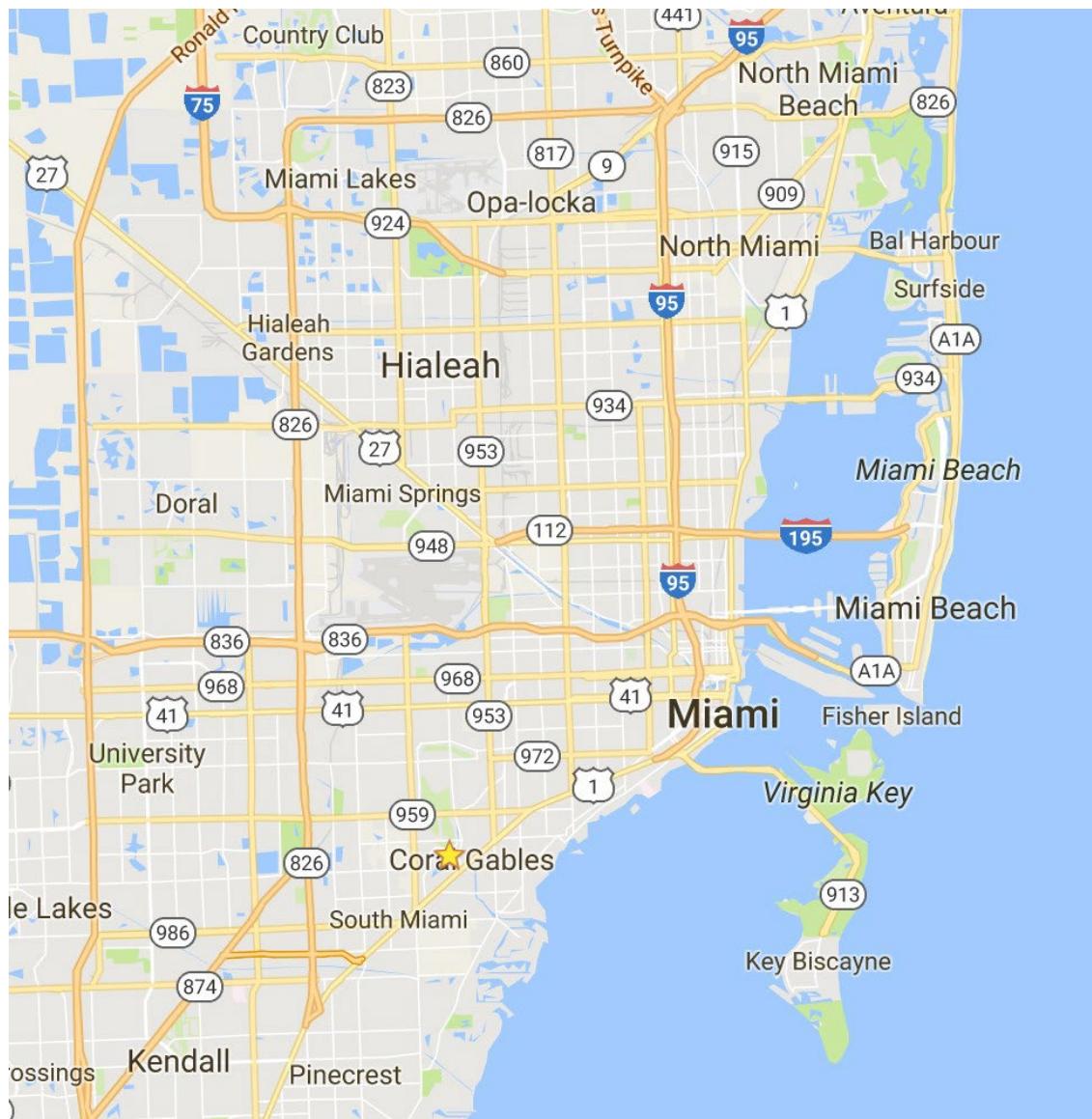
# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

Today...

# Graphs



facebook



Courtesy of Paul Butler

# Flight map



- Graphs
- Paths
- Depth-first search
- Breadth-first search

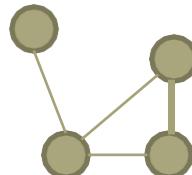
# Graphs

- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional

- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional
- Trees are a *subset* of graphs

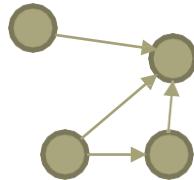
- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional
- Graphs are used **EXTENSIVELY** throughout CS

# Some definitions

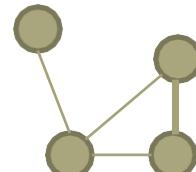


Undirected graph

# Some definitions

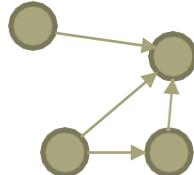


Directed graph

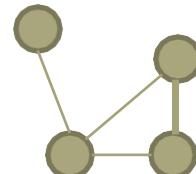


Undirected graph

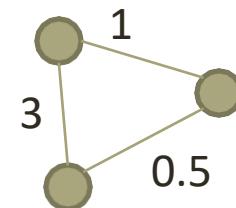
# Some definitions



Directed graph

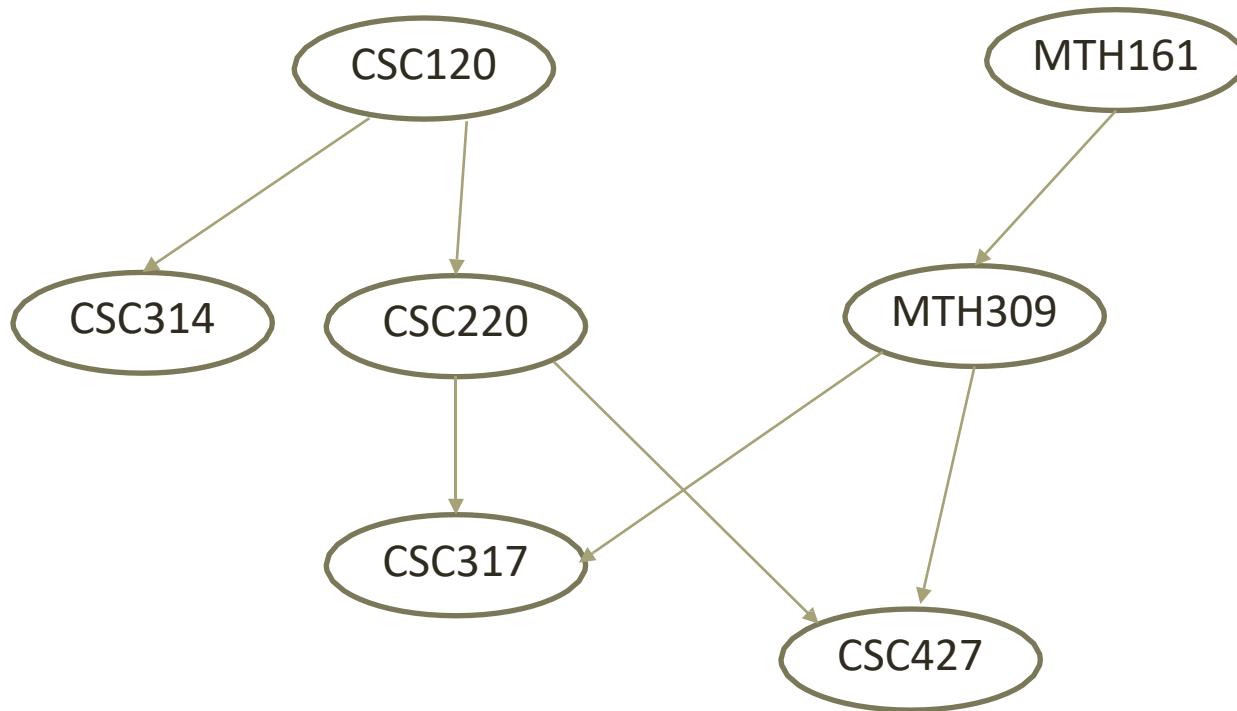


Undirected graph



Weighted graph

# What makes this a graph and not a tree?



- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    . . .  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

- The order in which neighbors appear in the list is unspecified
  - A different order still make the same graph!

# Paths

# Path finding

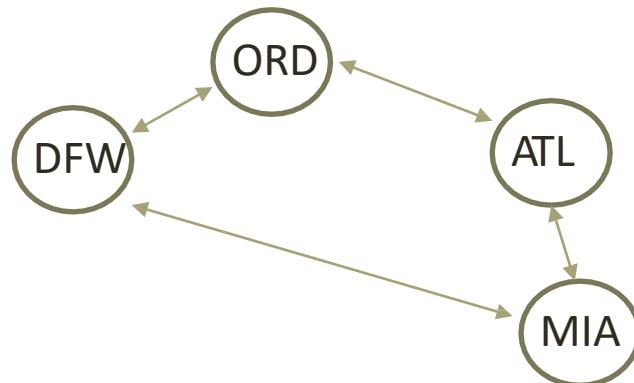
- A **path** is a sequence of nodes with
  - a start-point
  - and an end-point
- ...such that the end-point can be reached through a series of nodes from the start-point

# Path finding

- A **path** is a sequence of nodes with
  - a start-point
  - and an end-point
- ...such that the end-point can be reached through a series of nodes from the start-point

MIA — ATL — ORD

- There is *no direct path*

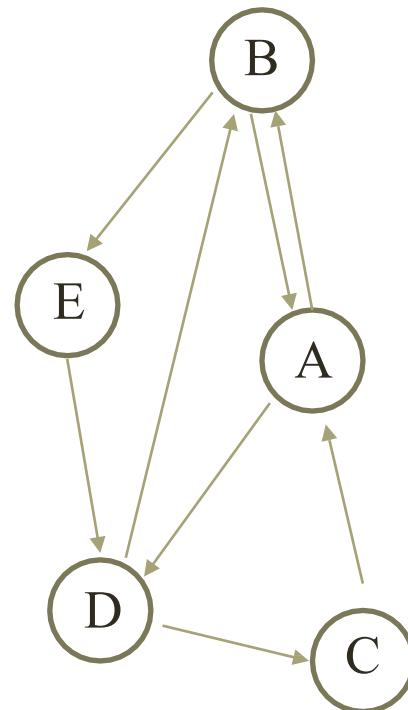


# Cycles

- A cycle in a graph is a path from a node back to itself

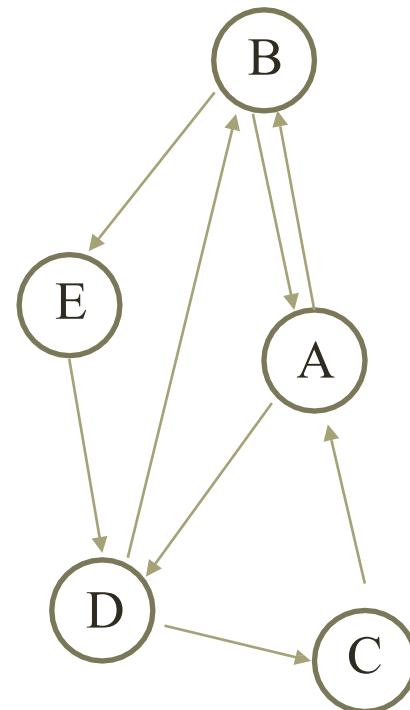
B — E — D — B

- While traversing a graph, special care must be taken to avoid cycles, otherwise what?
- Can trees have cycles?



# Path finding

- There may be more than one path from one node to another
- We are often interested in the *path length*
- Finding the shortest (or cheapest) path between two nodes is a common graph operation

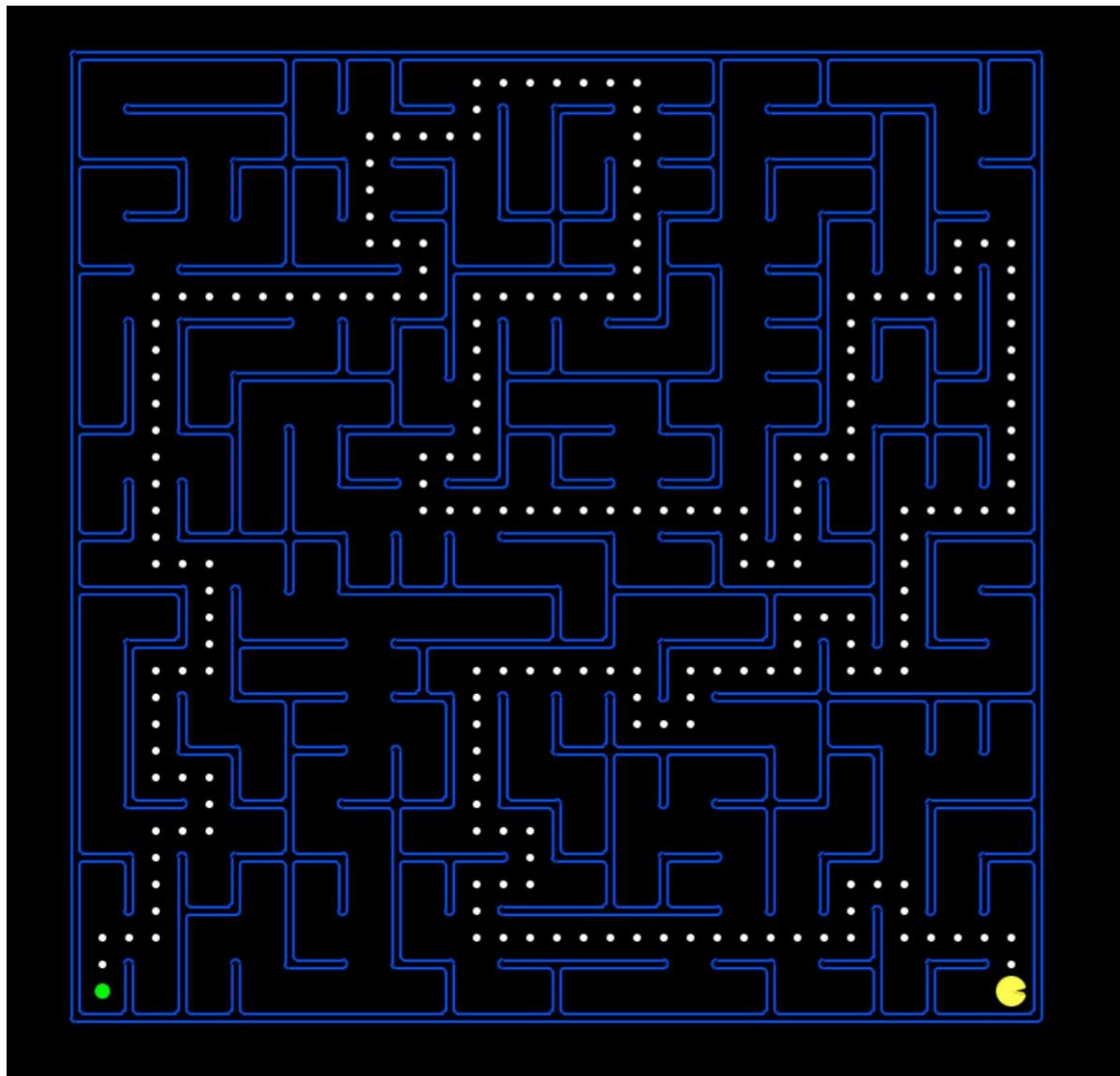


- Any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph

- Any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph
- ....and solved with pathfinding!

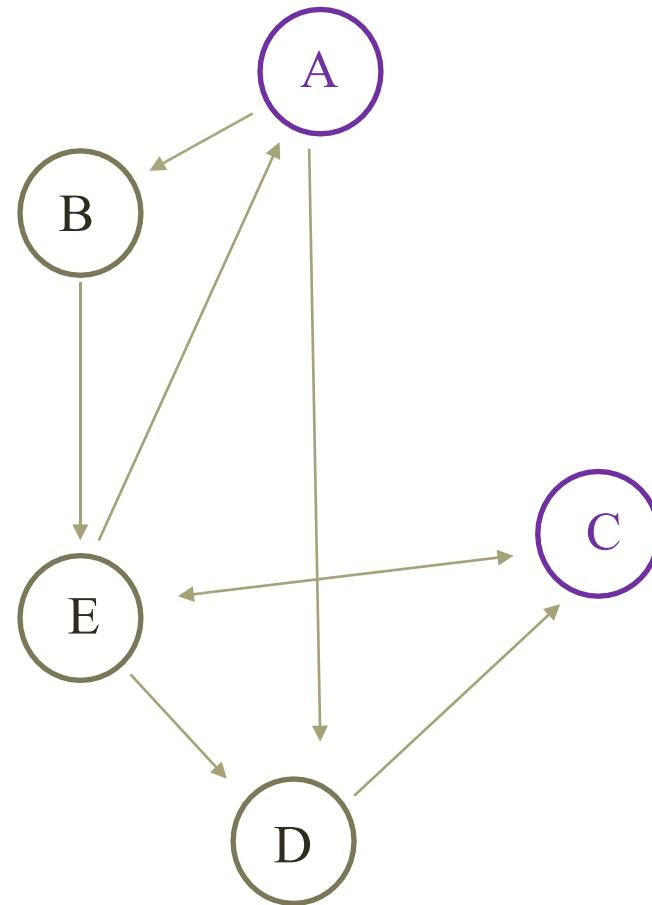
# Example

- In games, moving a character around a space
- Character finds the shortest path from its current location to the destination
  - Not always a straight line!
- Terrain is represented as a graph
  - Every non-obstacle spot on the terrain is a node
  - Nodes are connected to adjacent nodes
  - Navigating a maze...



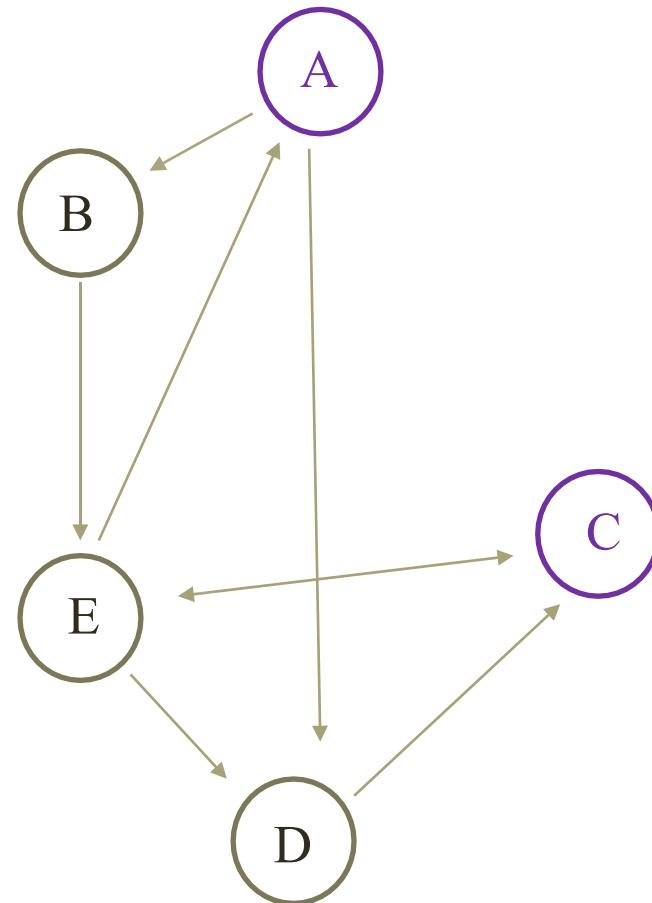
- Depth-first search — DFS
- Breadth-first search — BFS
- If there exists a path from one node to another these algorithms will find it
  - The nodes on this path are the steps to take to get from start point to the end point
- If multiple such paths exist, the algorithms may find different ones

We want to find a path from A to C



# Depth-first search

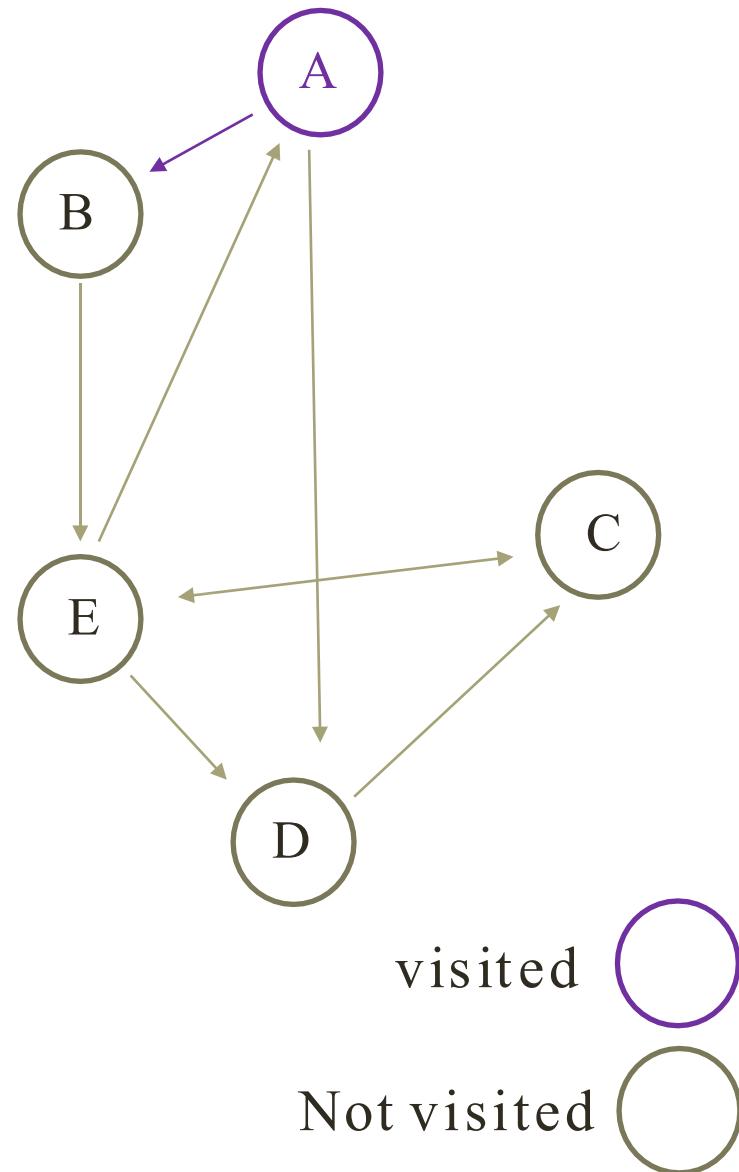
We want to find a path from A to C



We want to find a path from A to C

SO... Start from A,  
traverse its first edge, save  
where we came from, and  
recurse

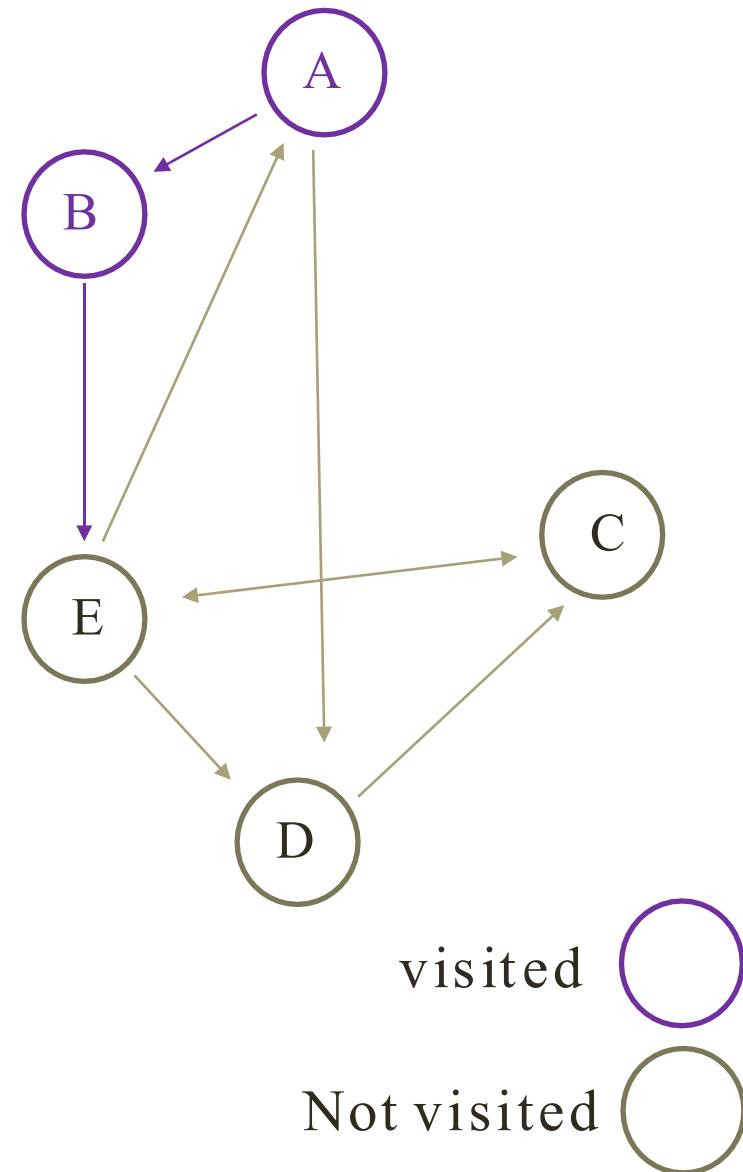
A.visited = true  
B.cameFrom = A



We want to find a path from A to C

Traverse the first unvisited node in the edge list recursively, save where we came from

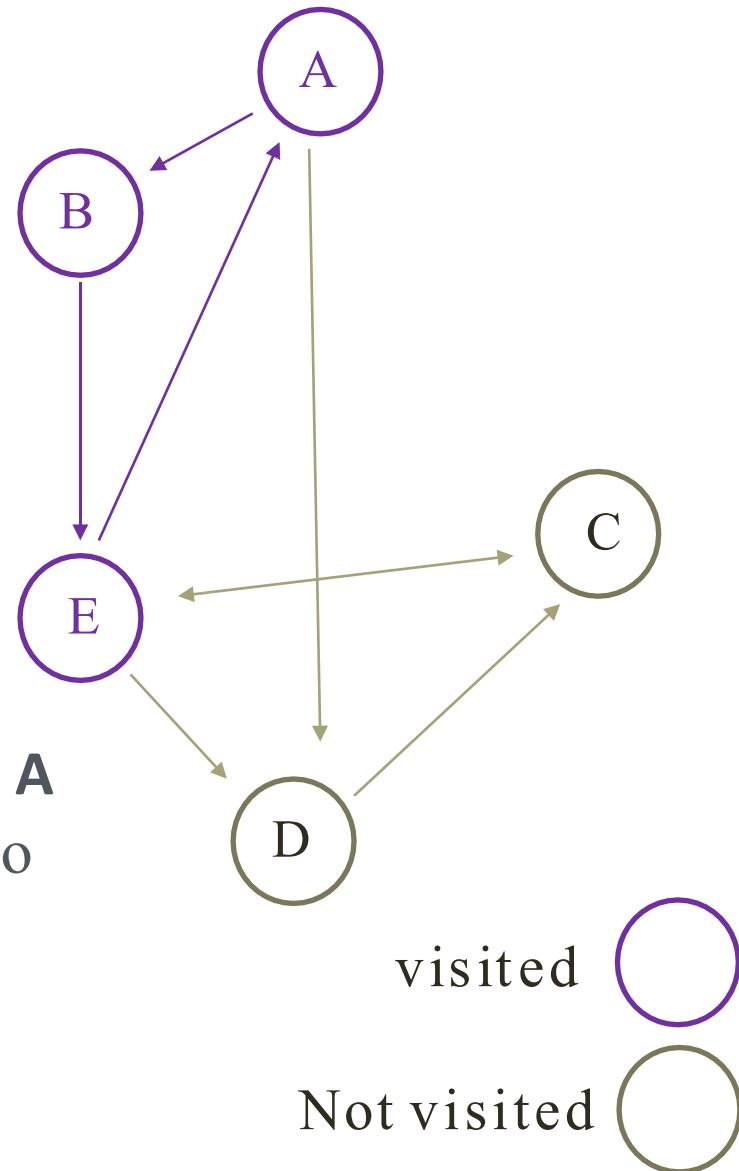
B.visited = true  
E.cameFrom = B



Traverse the first unvisited node in the edge list recursively, save where we came from

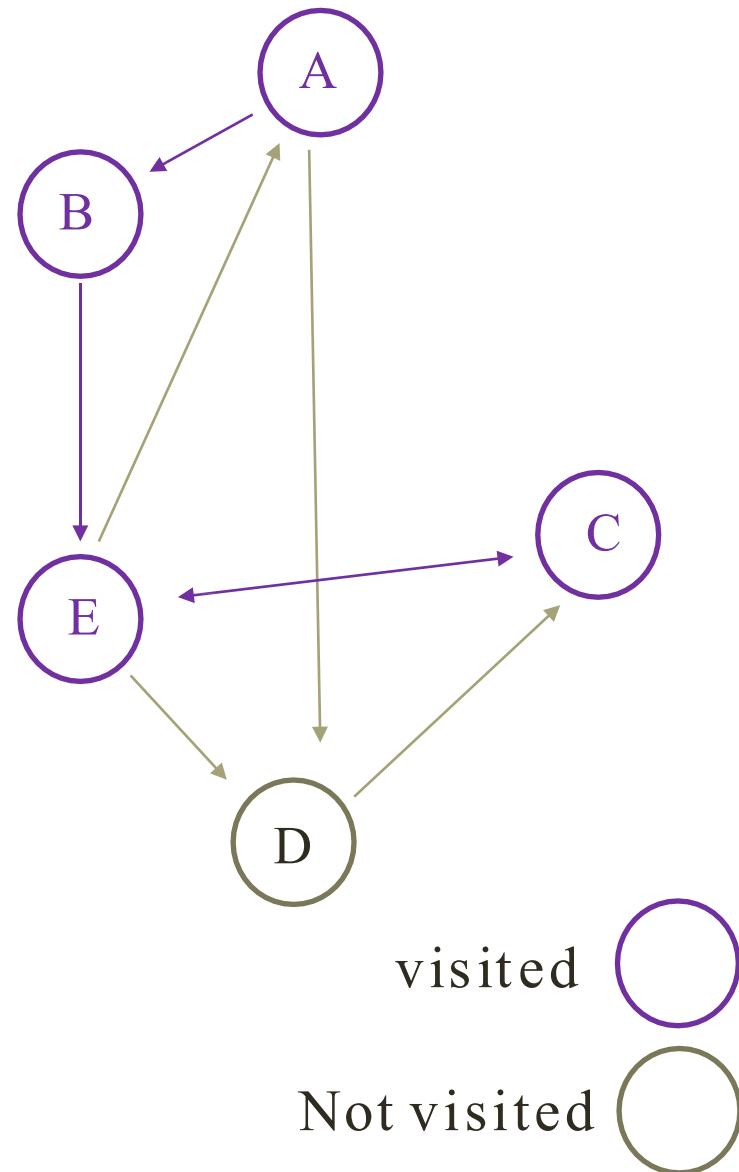
E.visited = true

Look at the first edge; node **A** has already been visited, so skip



Look at next edge; C has not been visited yet

C.cameFrom = E



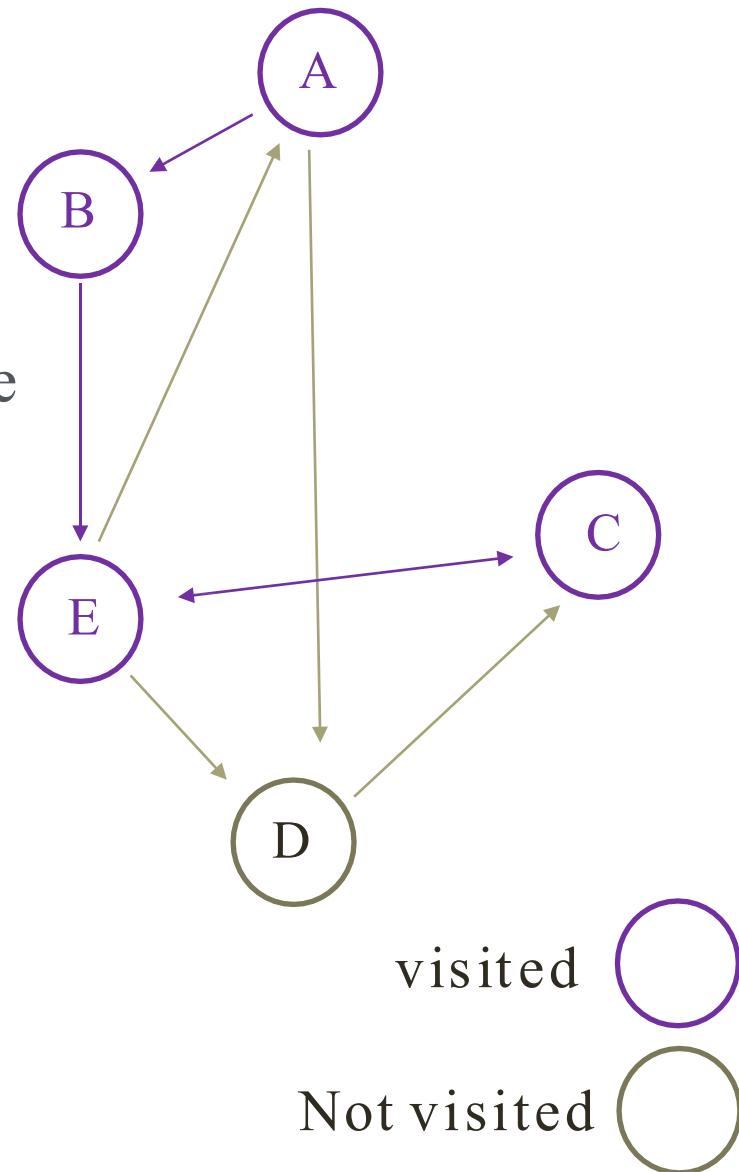
Node C is our goal. we are done!

C.visited = true

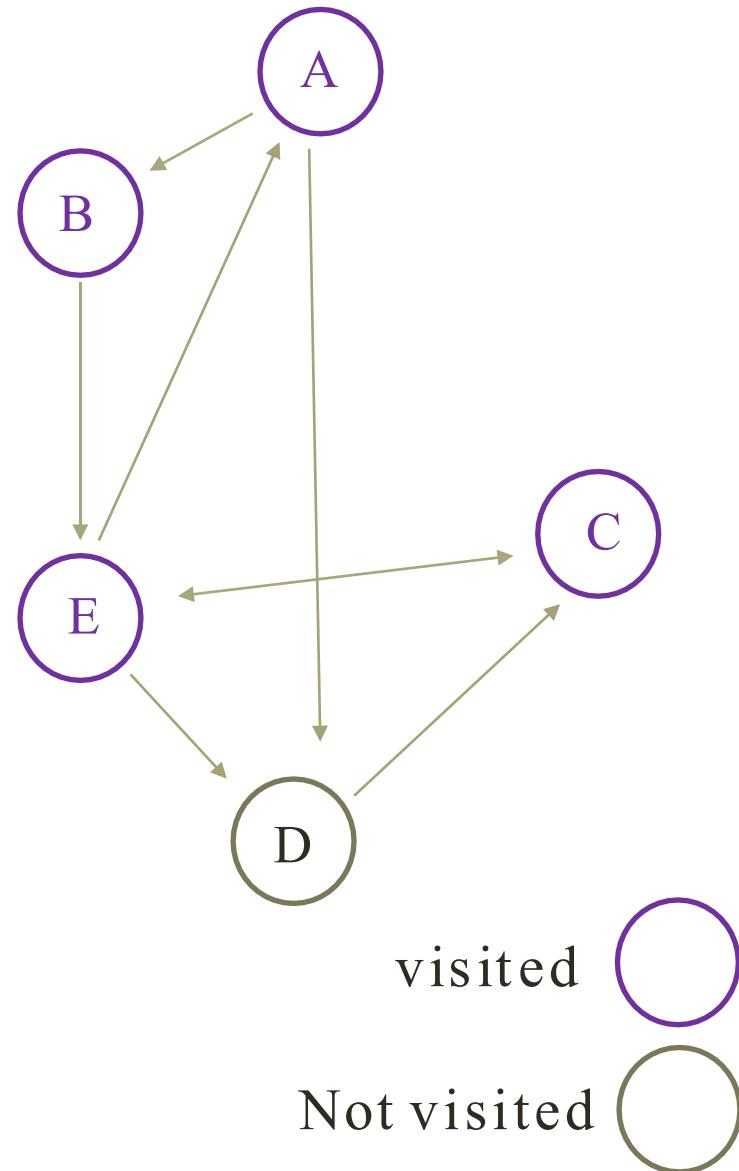
Follow each node's  
**cameFrom** to reconstruct the  
path

C.cameFrom = E,  
E.cameFrom = B,  
B.cameFrom = A

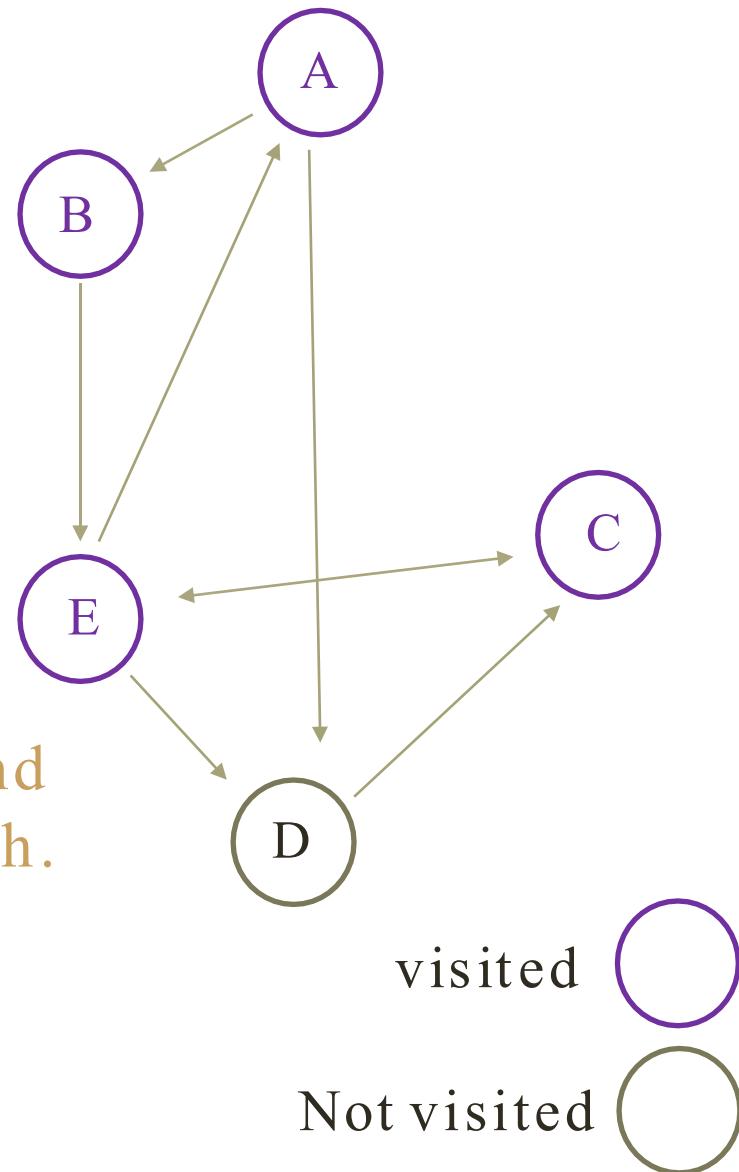
Path: A —B —E —C



Is there a better (shorter) path from A to C?



Is there a better (shorter) path from A to C?



```
DFS(Node curr, Node goal)
{
    curr.visited = true;
    if(curr.equals(goal))
        return;
    for(Node next : curr.neighbors)
        if(!next.visited)
        {
            next.cameFrom = curr;
            DFS(next, goal);
        }
}
// path is now saved in nodes' .cameFrom
```

```
1 // Java program to print DFS traversal from a given given graph
2 import java.io.*;
3 import java.util.*;
4
5 // This class represents a directed graph using adjacency list
6 // representation
7 class Graph
8 {
9     private int v; // No. of vertices
10
11    // Array of lists for Adjacency List Representation
12    private LinkedList<Integer> adj[];
13
14    // Constructor
15    Graph(int v)
16    {
17        this.v = v;
18        adj = new LinkedList[v];
19        for (int i=0; i<v; ++i)
20            adj[i] = new LinkedList();
21    }
22
23    //Function to add an edge into the graph
24    void addEdge(int v, int w)
25    {
26        adj[v].add(w); // Add w to v's list.
27    }
28
29    // A function used by DFS
30    void DFSUtil(int v,boolean visited[])
31    {
32        // Mark the current node as visited and print it
33        visited[v] = true;
34        System.out.print(v+" ");
35
36        // Recur for all the vertices adjacent to this vertex
37        Iterator<Integer> i = adj[v].listIterator();
38        while (i.hasNext())
39        {
40            int n = i.next();
41            if (!visited[n])
42                DFSUtil(n, visited);
43        }
44    }
}
```

```
45
46 // The function to do DFS traversal. It uses recursive DFSUtil()
47 void DFS(int v)
48 {
49     // Mark all the vertices as not visited(set as
50     // false by default in java)
51     boolean visited[] = new boolean[V];
52
53     // Call the recursive helper function to print DFS traversal
54     DFSUtil(v, visited);
55 }
56
57 public static void main(String args[])
58 {
59     Graph g = new Graph(4);
60
61     g.addEdge(0, 1);
62     g.addEdge(0, 2);
63     g.addEdge(1, 2);
64     g.addEdge(2, 0);
65     g.addEdge(2, 3);
66     g.addEdge(3, 3);
67
68     System.out.println("Following is Depth First Traversal "+
69                     "(starting from vertex 2)");
70
71     g.DFS(2);
72 }
73 }
74 // This code is contributed by Aakash Hasija
75
```

# Summary

- Look at the first edge going out of the start node
- Recursively search from the new node
- Upon returning, take the next edge
- If no more edges, return

# Summary

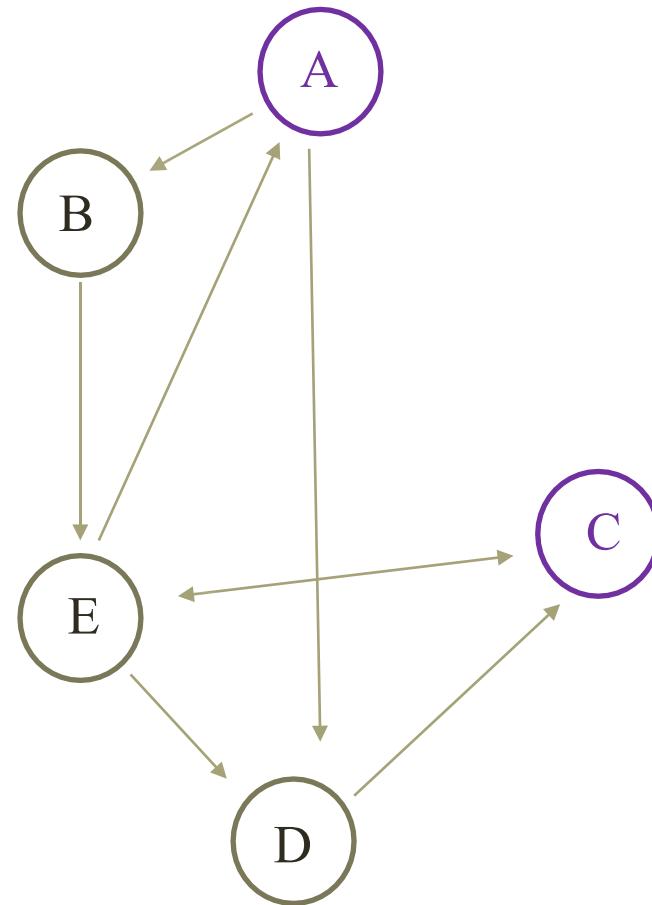
- Look at the first edge going out of the start node
  - Recursively search from the new node
  - Upon returning, take the next edge
  - If no more edges, return
- 
- When visiting a node, mark it as visited
    - So we don't get stuck in a cycle
    - Skip already visited nodes during traversal

# Summary

- Look at the first edge going out of the start node
  - Recursively search from the new node
  - Upon returning, take the next edge
  - If no more edges, return
- 
- When visiting a node, mark it as visited
    - So we don't get stuck in a cycle
    - Skip already visited nodes during traversal
  - For each node visited, save a reference to the node where we came from to reconstruct the path

# Breadth-first search

We want to find a path from A to C

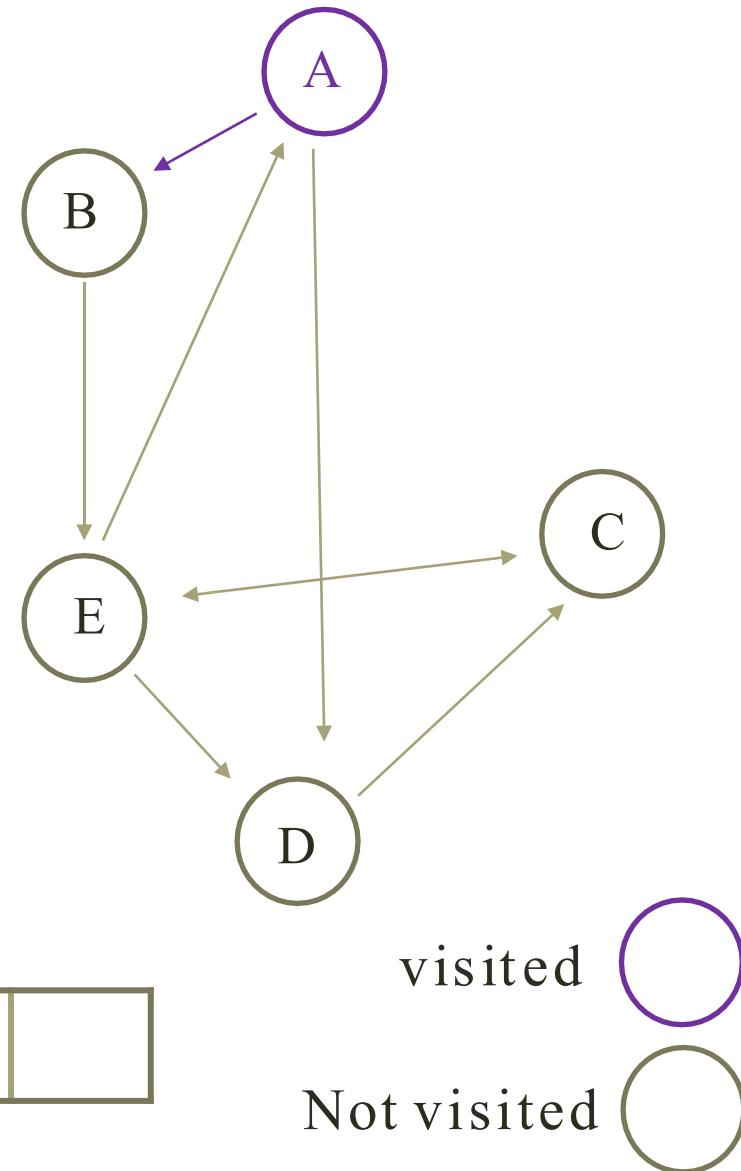


We want to find a path from A to C

Mark and enqueue the start node A

A.visited = true

Queue



visited



Not visited

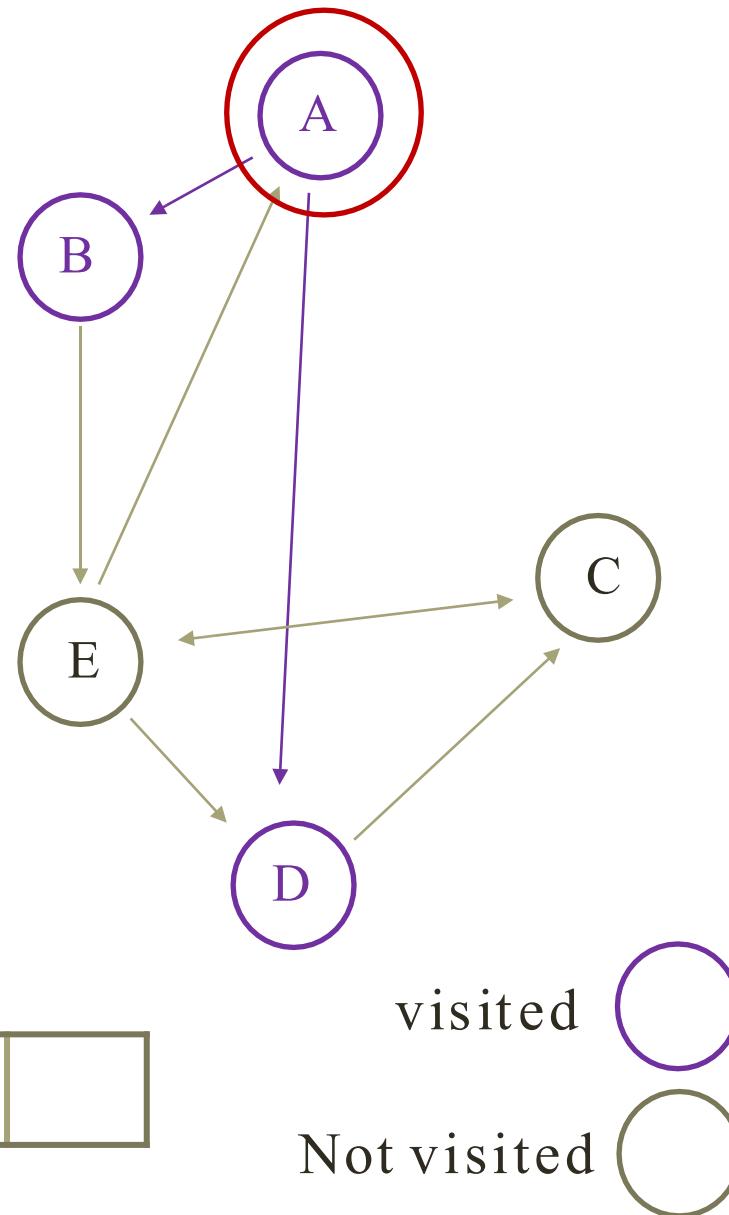


Dequeue the first node in the queue (A)

Mark and enqueue A's unvisited neighbors

```
B.cameFrom = A  
D.cameFrom = A  
B.visited = true  
D.visited = true
```

Queue [ B | D | ]

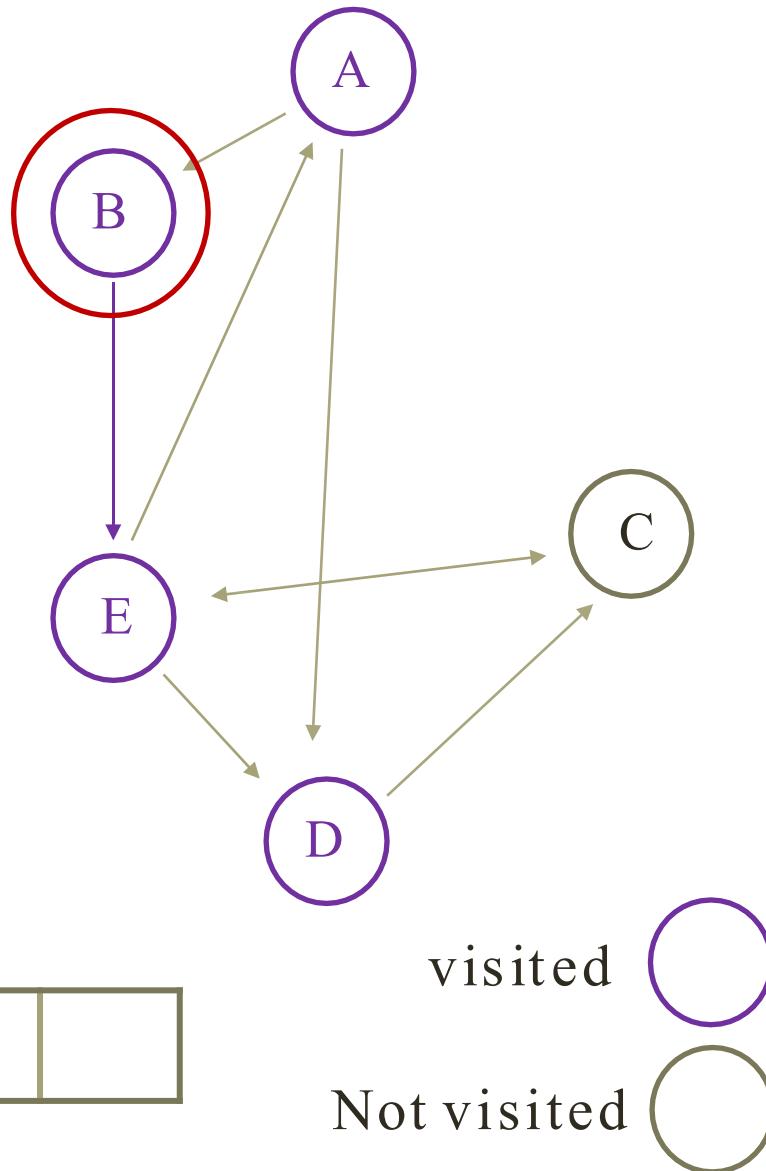


Dequeue the first node in the queue (B)

Mark and enqueue B's unvisited neighbors

E.cameFrom = B  
E.visited = true

Queue [D | E | ]

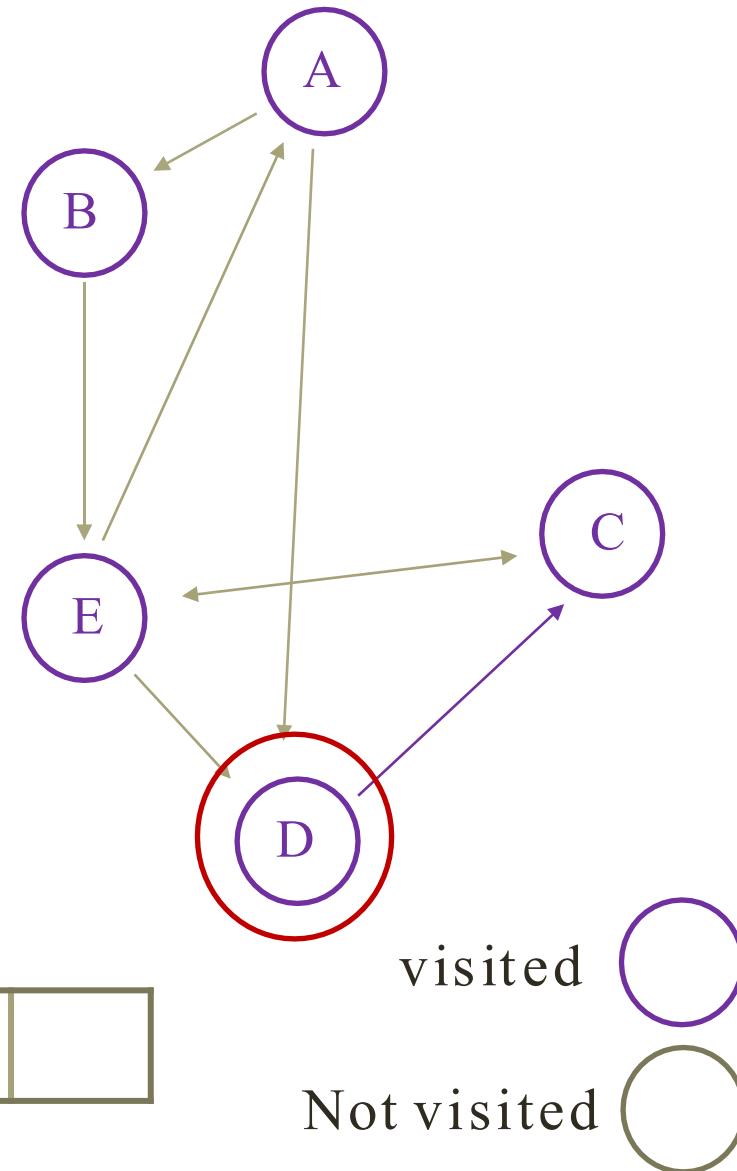


Dequeue the first node in the queue (D)

Mark and enqueue D's  
unvisited neighbors

C.cameFrom = D  
C.visited = true

Queue [E | C | ]

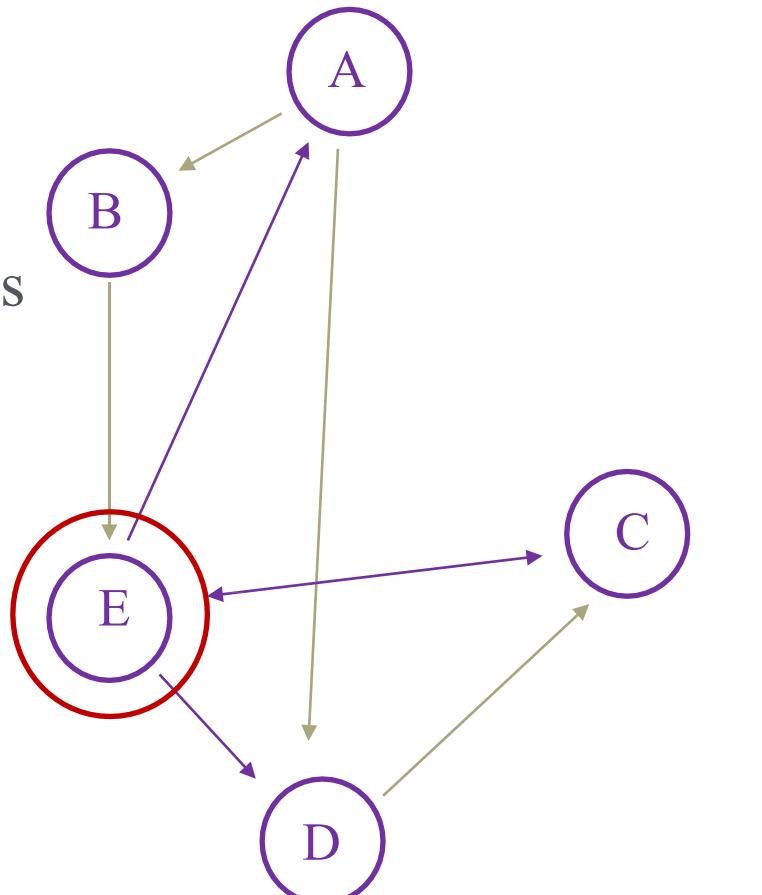


Dequeue the first node in the queue (E)

Mark and enqueue E's  
unvisited visited neighbors

(no unvisited neighbors!)

Queue



visited

Not visited

Dequeue the first node in the queue (C)

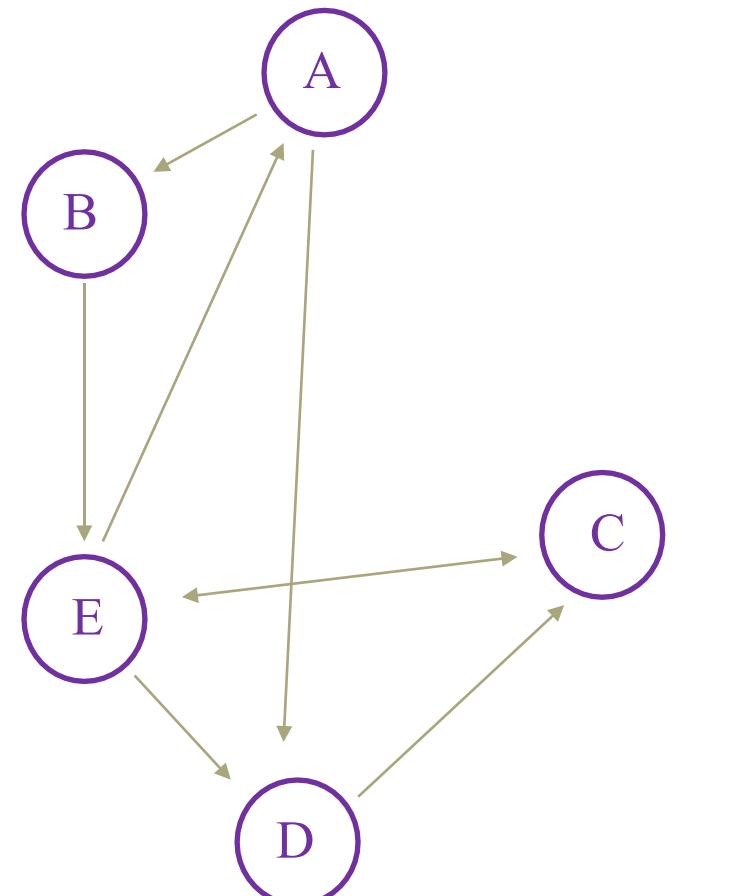
C is the goal! Reconstruct  
the path visited  
with **cameFrom** references

C.cameFrom = D

D.cameFrom = A

Path: A —D —C

Queue



visited

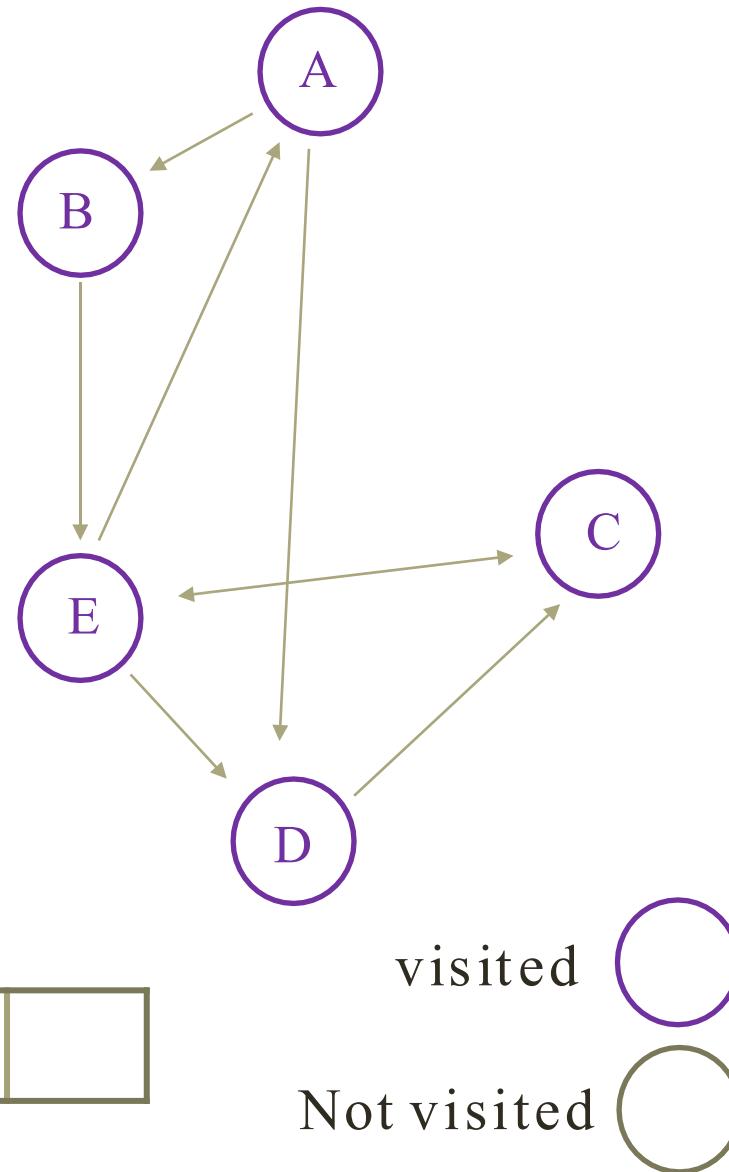


Not visited



Is this the shortest path?

Path: A — D — C



Queue



Not visited

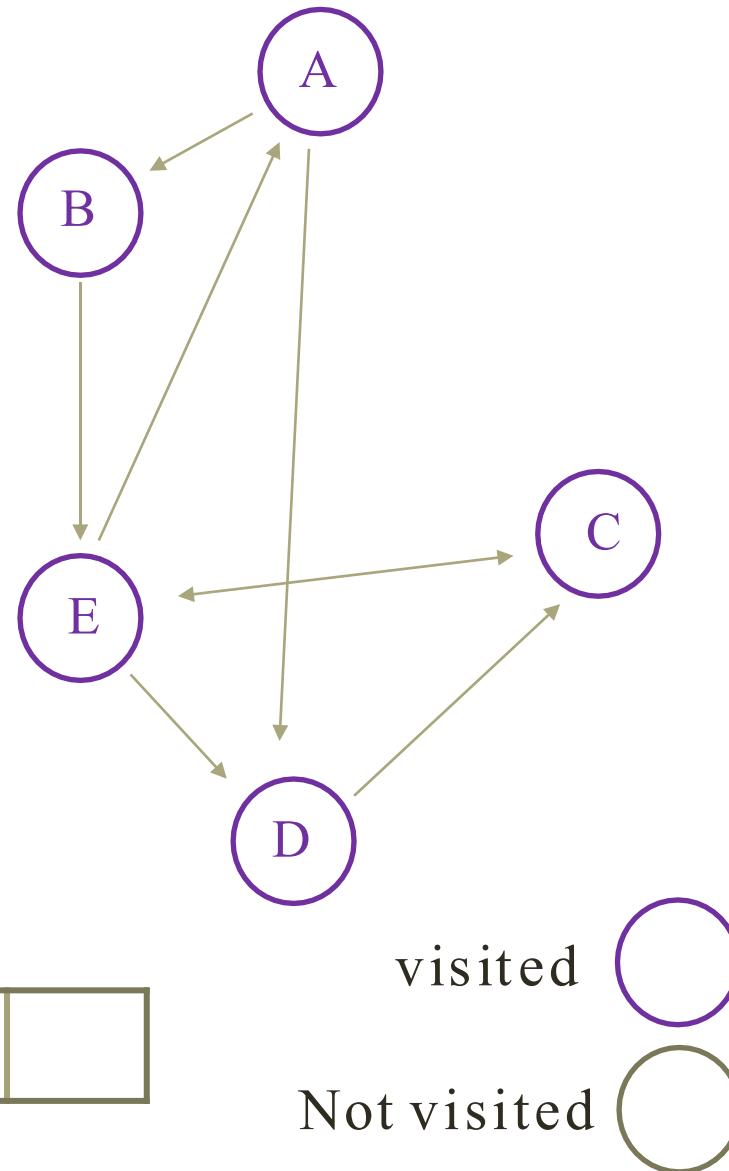
Is this the shortest path?

Path: A — D — C

BFS visits nodes closest to the start-point first

Therefore, the first path found is the shortest path (closest to the start node)

Queue



```
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)
    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return
        for(Node next : curr.neighbors)
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
```

```
1 // Java program to print BFS traversal from a given source vertex.
2 // BFS(int s) traverses vertices reachable from s.
3 import java.io.*;
4 import java.util.*;
5
6 // This class represents a directed graph using adjacency list
7 // representation
8 class Graph
9 {
10     private int v; // No. of vertices
11     private LinkedList<Integer> adj[]; //Adjacency Lists
12
13     // Constructor
14     Graph(int v)
15     {
16         V = v;
17         adj = new LinkedList[v];
18         for (int i=0; i<v; ++i)
19             adj[i] = new LinkedList();
20     }
21
22     // Function to add an edge into the graph
23     void addEdge(int v,int w)
24     {
25         adj[v].add(w);
26     }
```

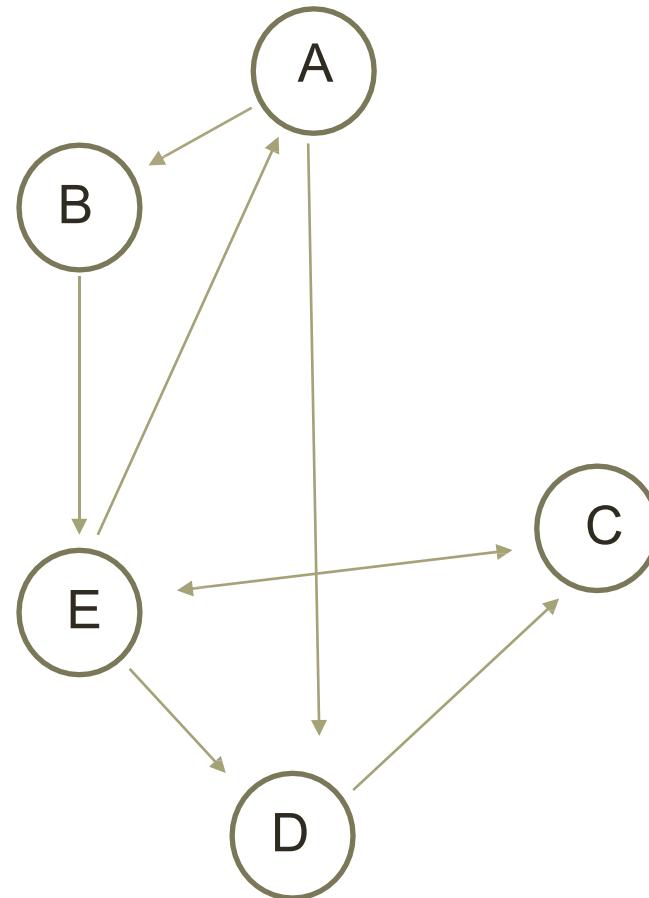
```
27
28 // prints BFS traversal from a given source s
29 void BFS(int s)
30 {
31     // Mark all the vertices as not visited(By default
32     // set as false)
33     boolean visited[] = new boolean[V];
34
35     // Create a queue for BFS
36     LinkedList<Integer> queue = new LinkedList<Integer>();
37
38     // Mark the current node as visited and enqueue it
39     visited[s]=true;
40     queue.add(s);
41
42     while (queue.size() != 0)
43     {
44         // Dequeue a vertex from queue and print it
45         s = queue.poll();
46         System.out.print(s+" ");
47
48         // Get all adjacent vertices of the dequeued vertex s
49         // If a adjacent has not been visited, then mark it
50         // visited and enqueue it
51         Iterator<Integer> i = adj[s].listIterator();
52         while (i.hasNext())
53         {
54             int n = i.next();
55             if (!visited[n])
56             {
57                 visited[n] = true;
58                 queue.add(n);
59             }
60         }
61     }
62 }
```

```
64 // Driver method to
65 public static void main(String args[])
66 {
67     Graph g = new Graph(4);
68
69     g.addEdge(0, 1);
70     g.addEdge(0, 2);
71     g.addEdge(1, 2);
72     g.addEdge(2, 0);
73     g.addEdge(2, 3);
74     g.addEdge(3, 3);
75
76     System.out.println("Following is Breadth First Traversal "+
77                         "(starting from vertex 2)");
78
79     g.BFS(2);
80 }
81 }
82 // This code is contributed by Aakash Hasija
83 }
```

- Instead of visiting deeper nodes first, visit shallower nodes first
  - Visit nodes closest to the start point first, gradually get further away
- Create an empty queue
- Put the starting node in the queue
- While the queue is not empty
  - Dequeue the current node
  - For each unvisited neighbor of the current node
    - *Mark the neighbor as visited*
    - *Put the neighbor into the queue*
- Notice it is not recursive... it just runs until the queue is empty!

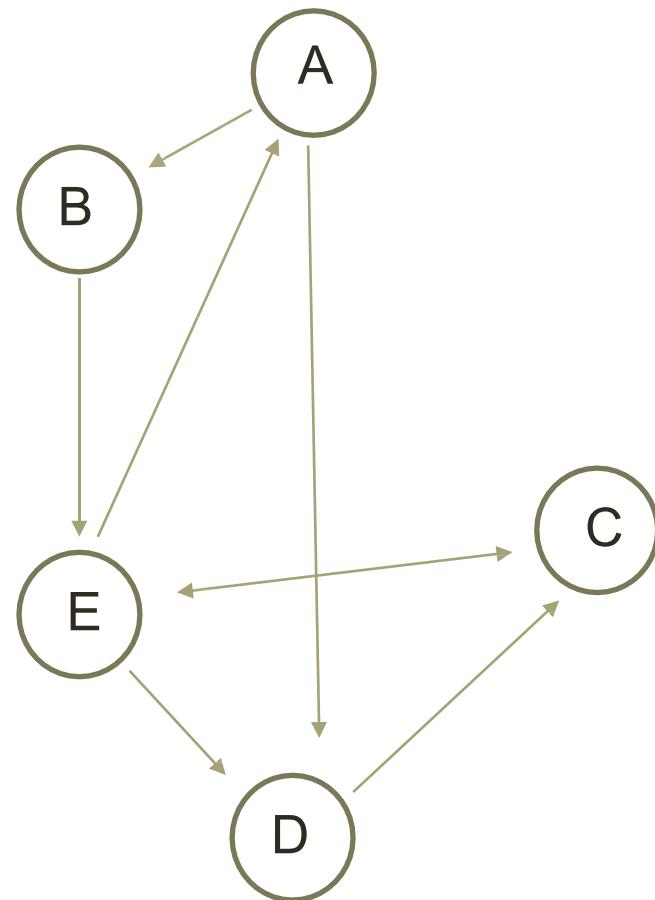
What path will BFS find from B to C?

- A) B E C
- B) B E A D C
- C) B E D C
- D) none



What path will DFS find from A to D?

- A) A B E D
- B) A D
- C) none
- D) this is a trick question



What is true of DFS, searching from a start node to a goal node?

- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a and b
- E) a and c
- F) a, b, and c

What is true of BFS, searching from a start node to a goal node?

- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a, and b
- F) a, and c

# Summary

- Depth-first search — DFS
- Breadth-first search — BFS
- If there exists a path from one node to another these algorithms will find it
  - The nodes on this path are the steps to take to get from point A to point B
- If multiple such paths exist, the algorithms may find different ones

Next time...

- Continue with graphs
- Quiz
- Start on your assignment early!

# Graphs

## CSC220|Computer Programming 2

Last Time...

# Heap sort

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- How?

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array ☺

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array 😊 → Why?!

# Pseudo code...

A = array to be sorted

H = create new heap

For each element n in A

    add n to H

While (H not empty)

    remove element from H

    add element back into A

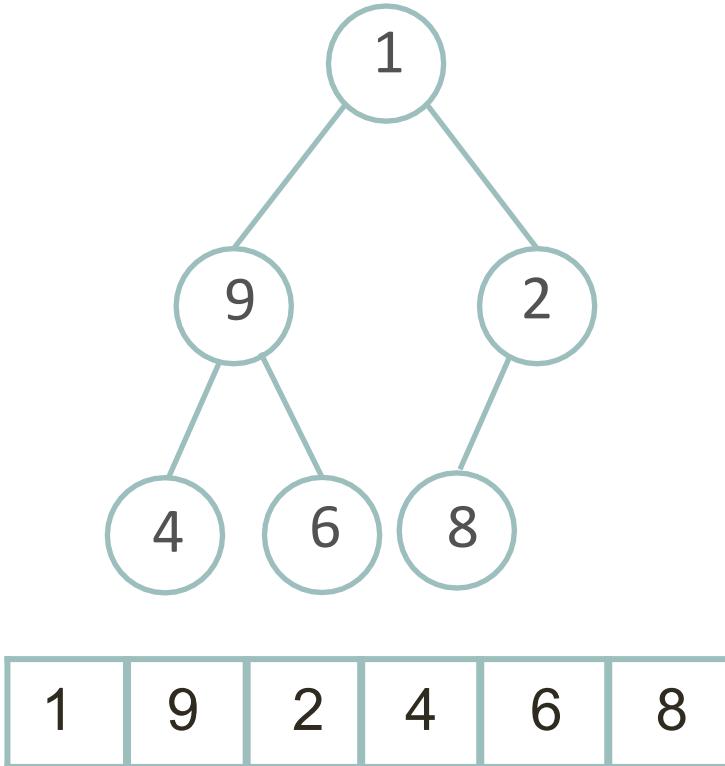
# In-place heap sort

- Use the original array itself as the heap
- The idea is to treat the array as an initially invalid heap and repair it into a proper heap
  - Bubbling elements into their proper position
- Remove elements from the heap and put them at the end of the array.
- min-heap gives you descending order and max-heap gives you ascending order

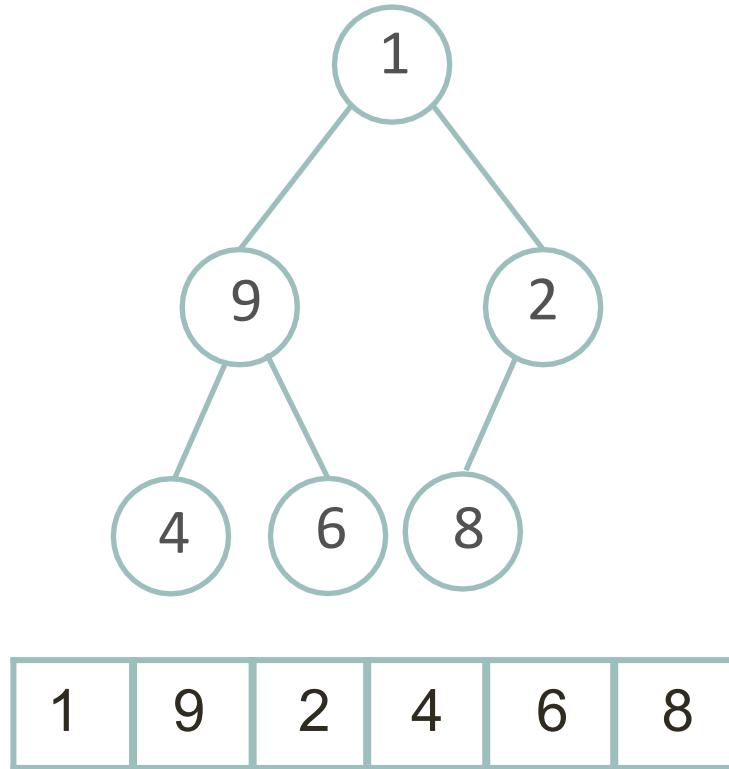
# In-place heap sort

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 9 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|

# In-place heap sort

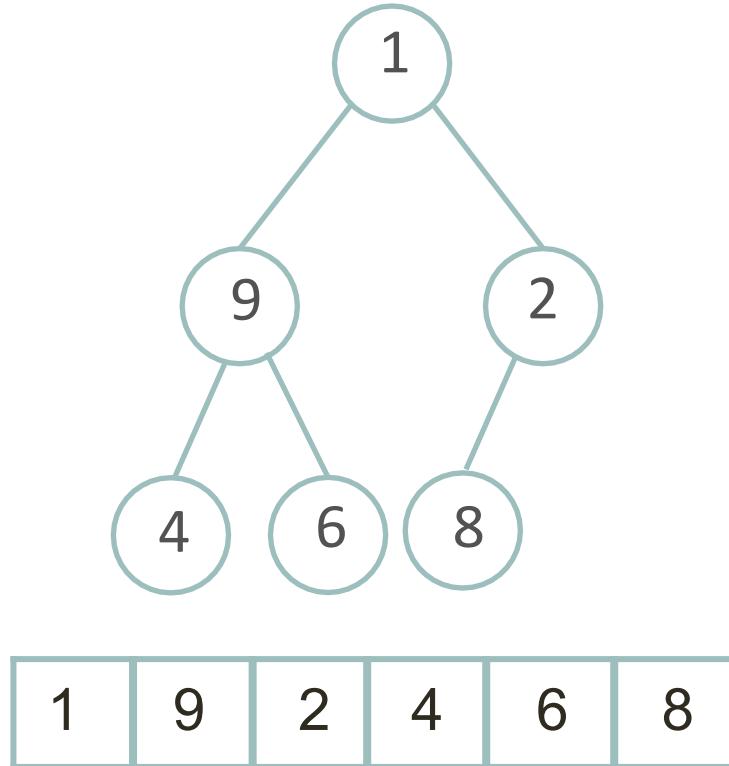


# In-place heap sort



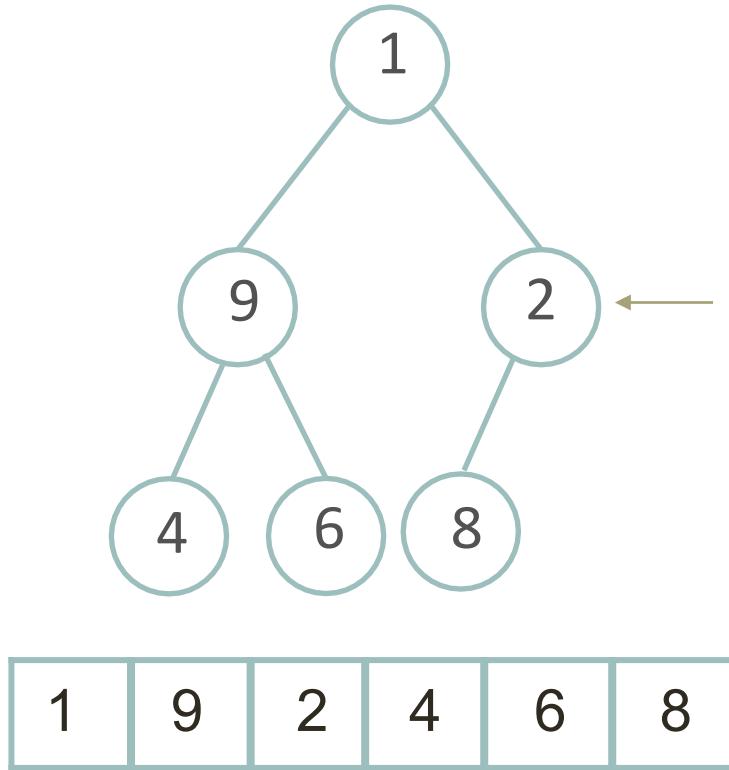
Not a heap → Need to *heapify*

# In-place heap sort



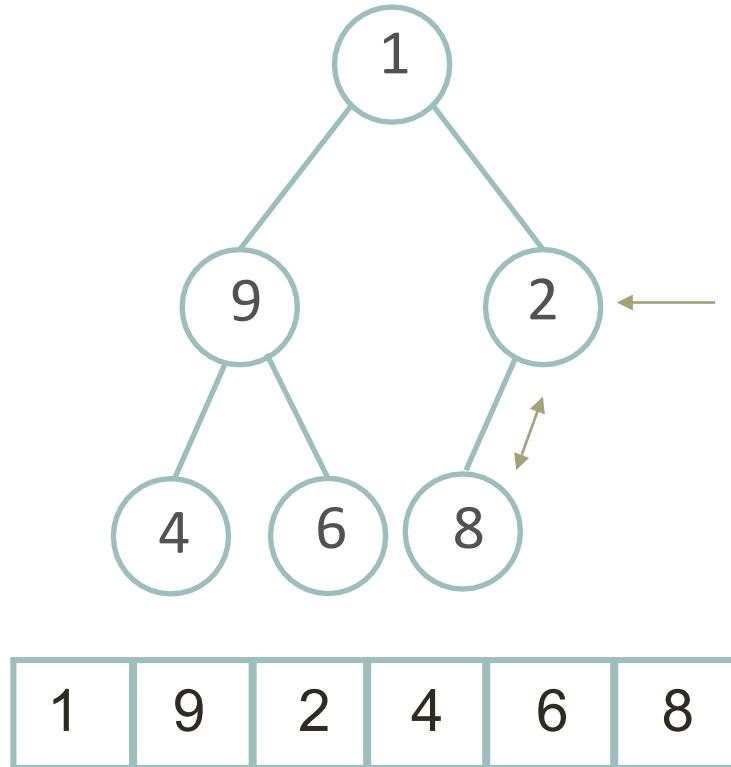
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



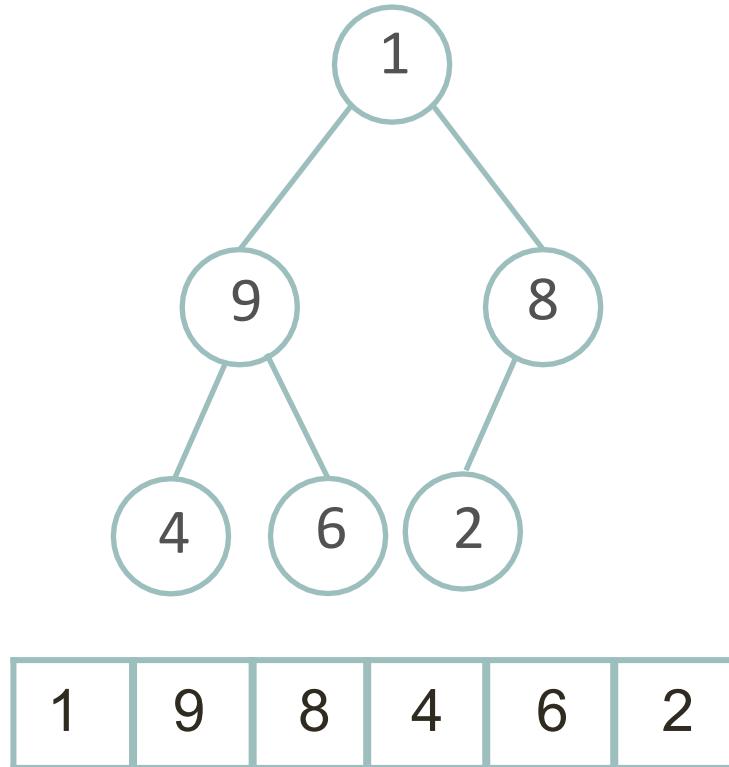
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



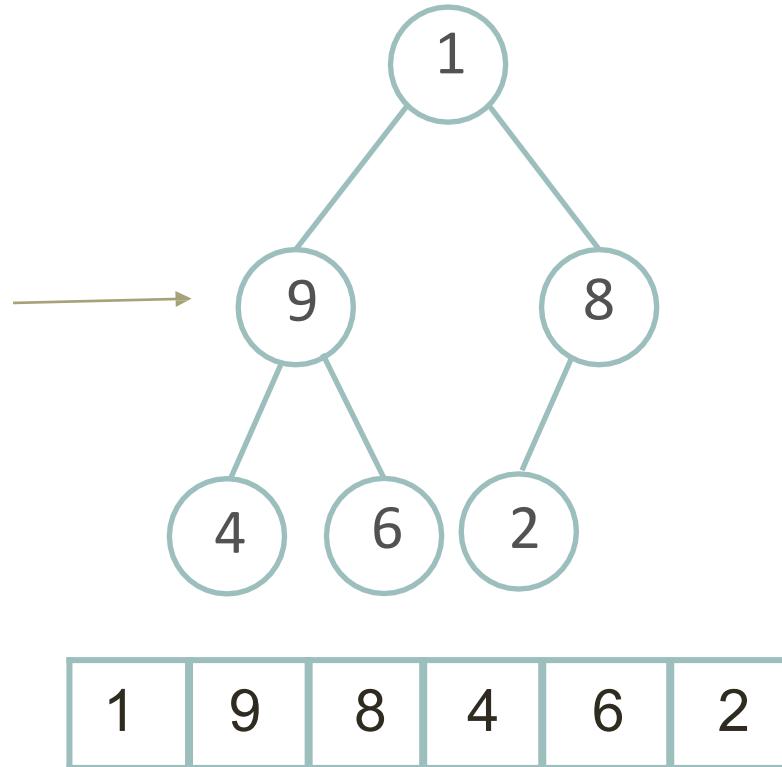
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



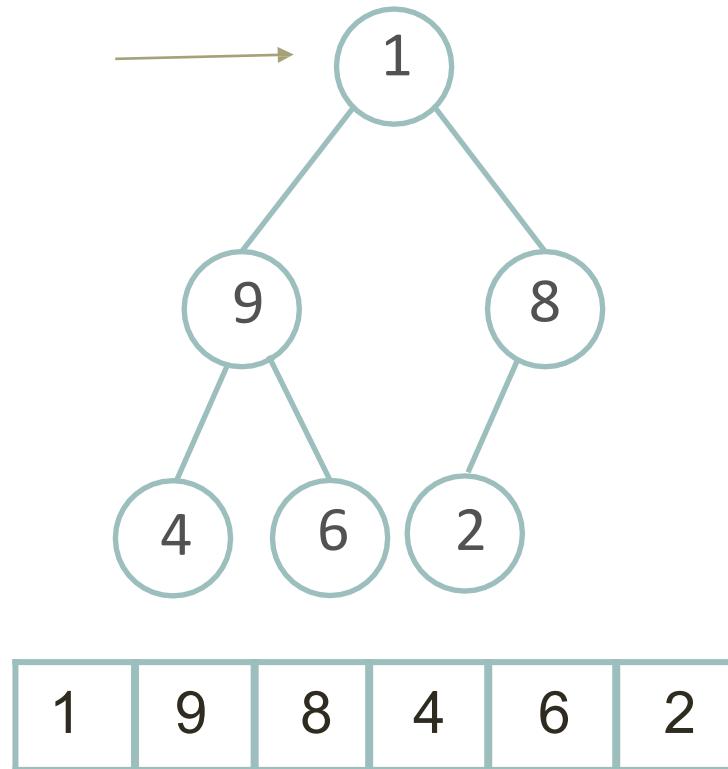
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



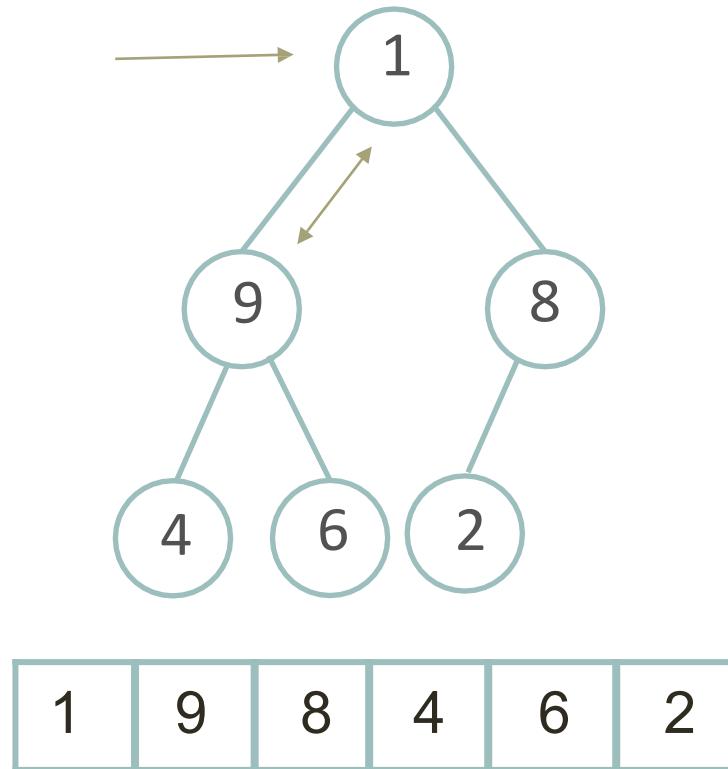
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



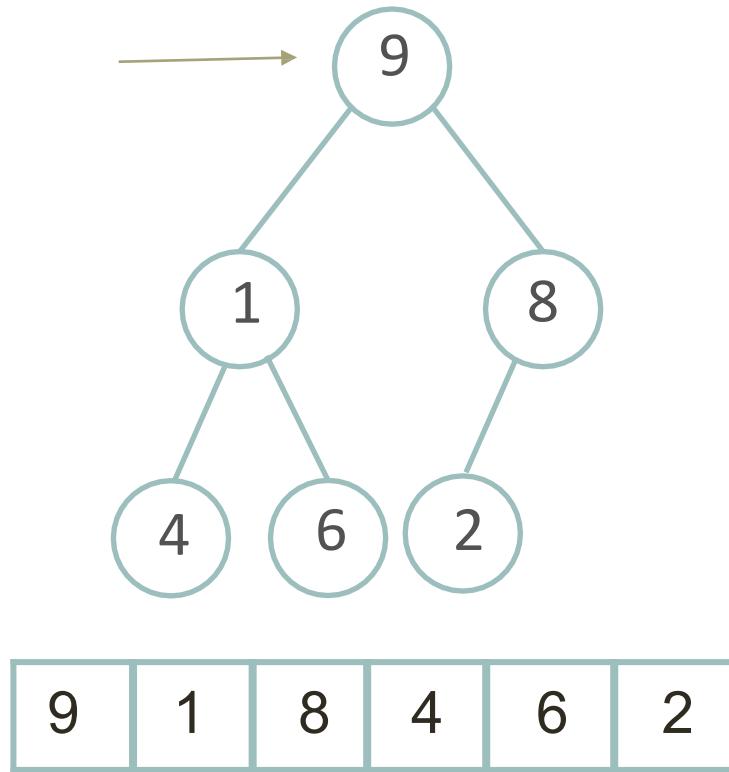
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



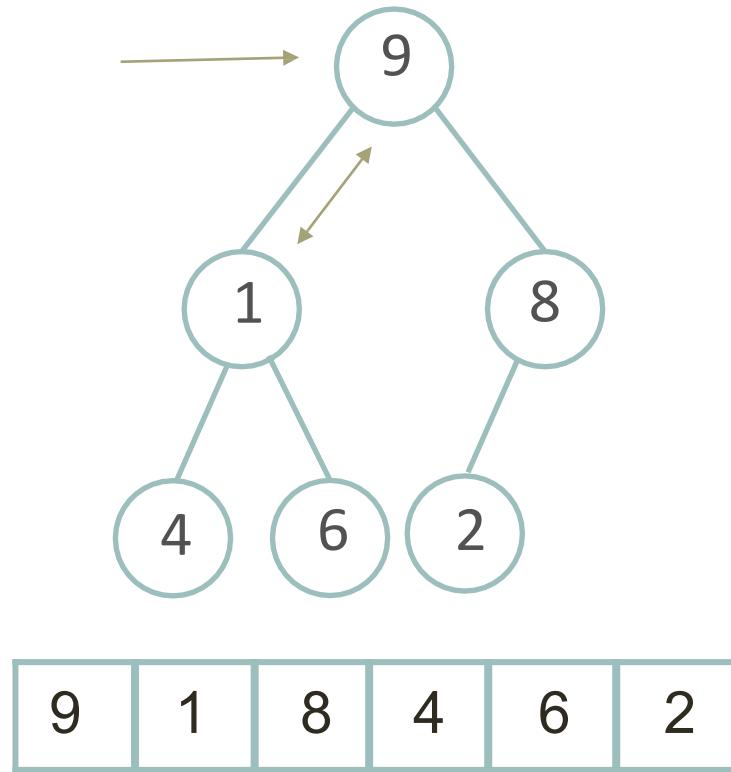
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



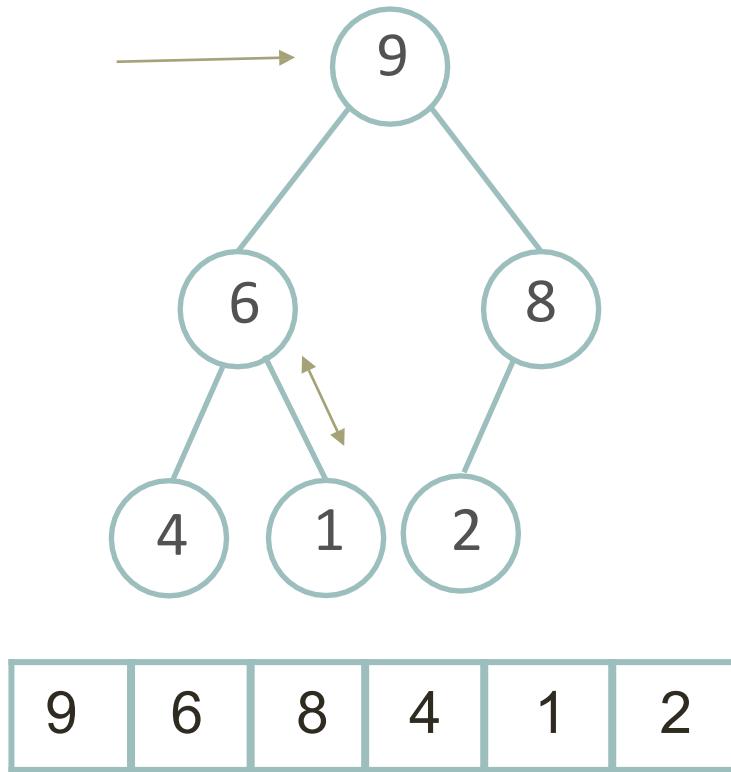
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

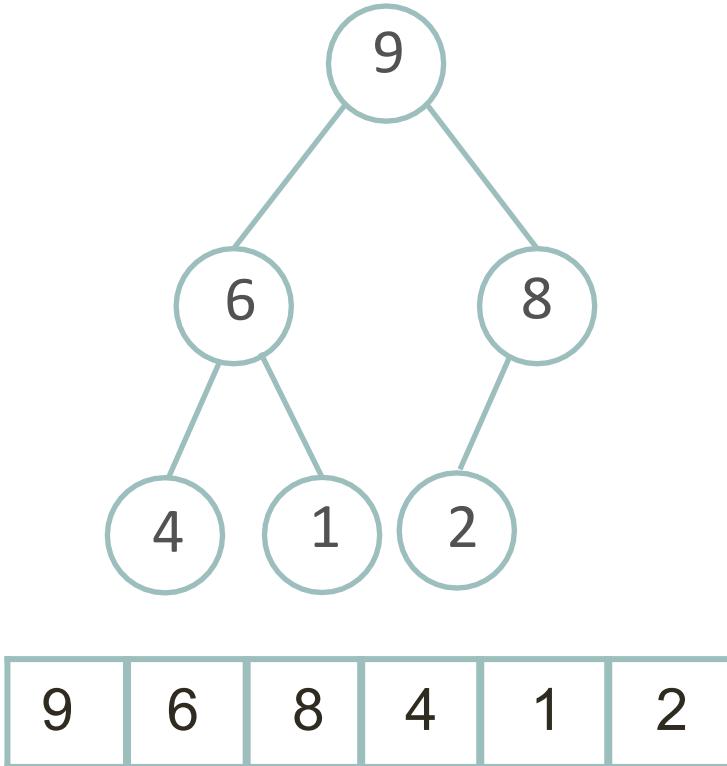


Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

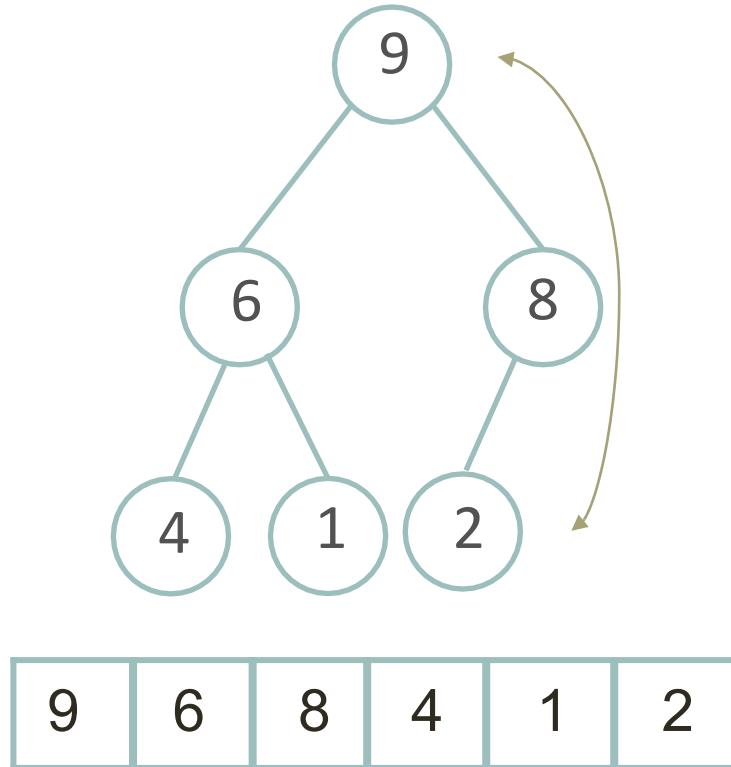


# In-place heap sort



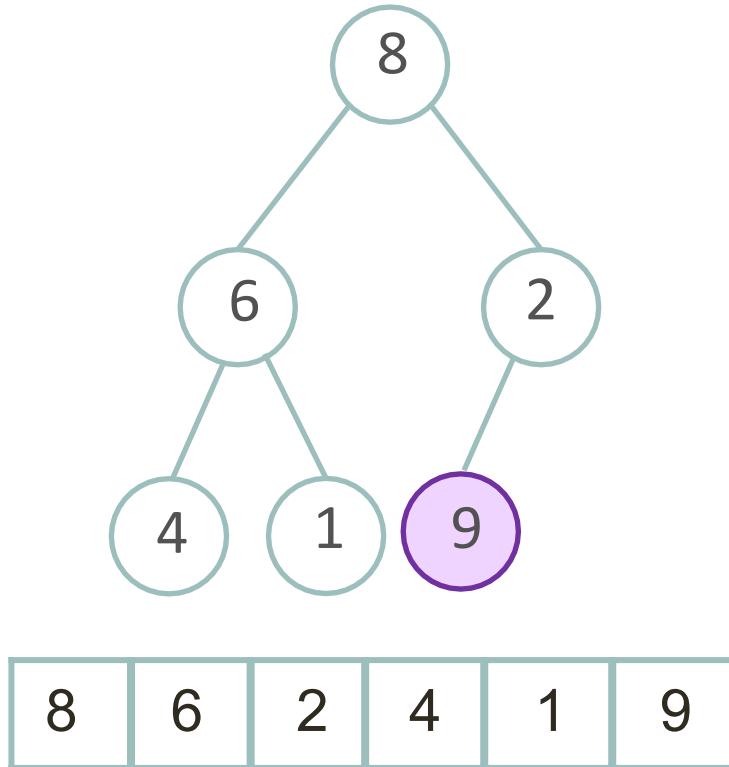
Not yet sorted array!

# In-place heap sort



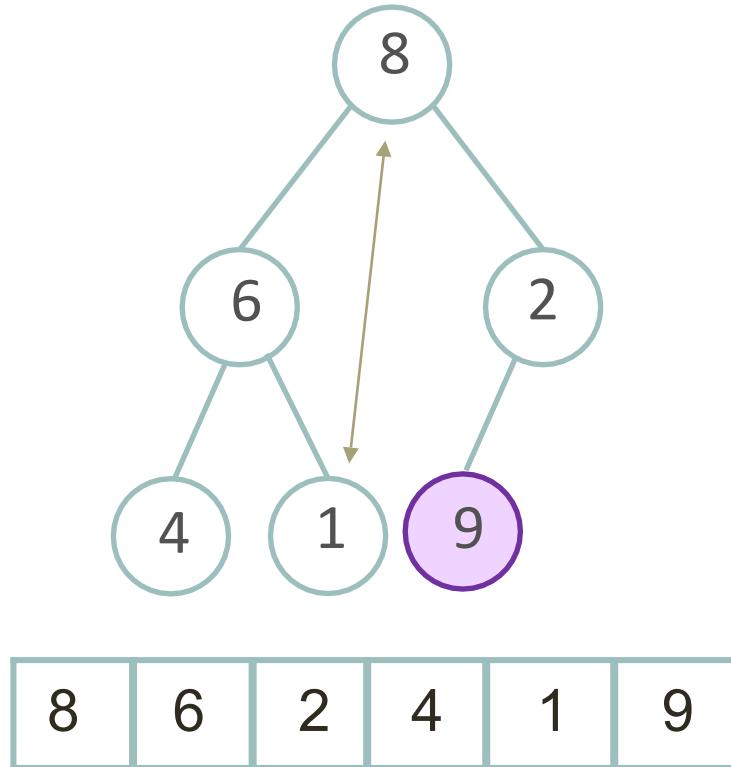
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



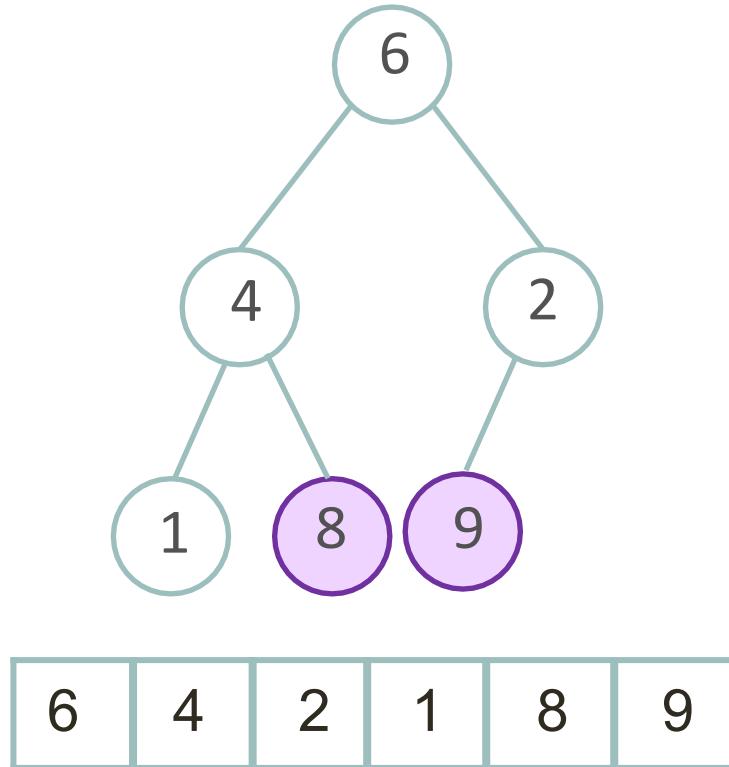
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



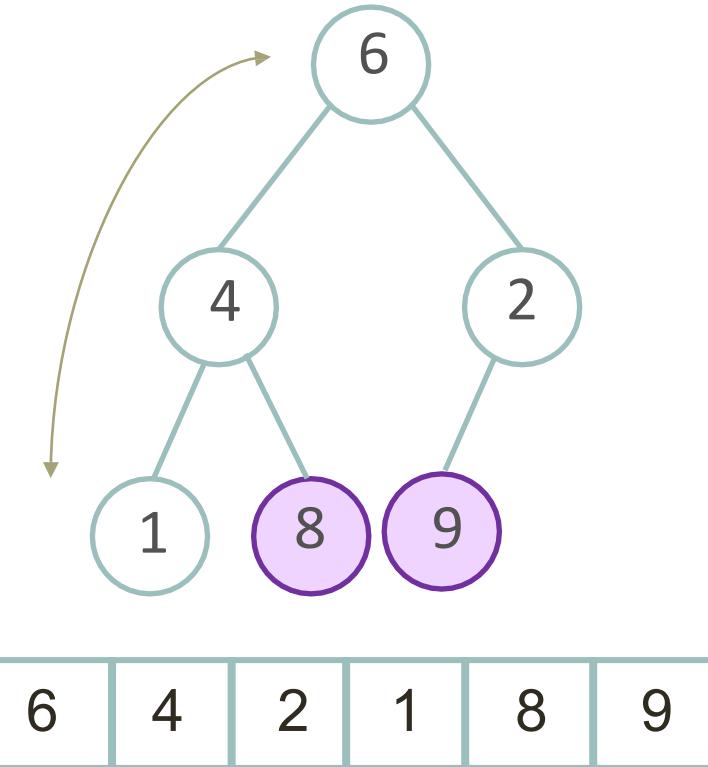
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



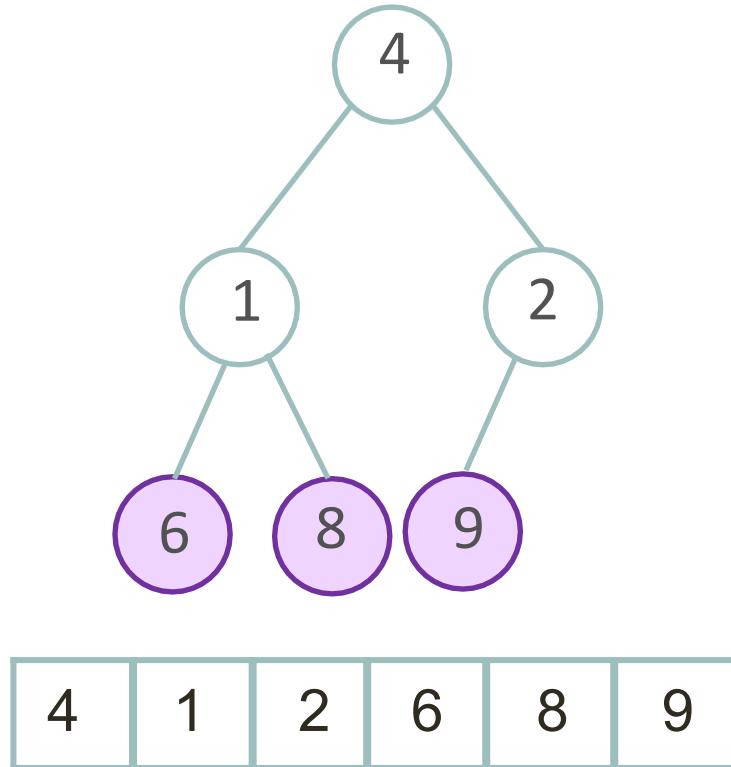
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



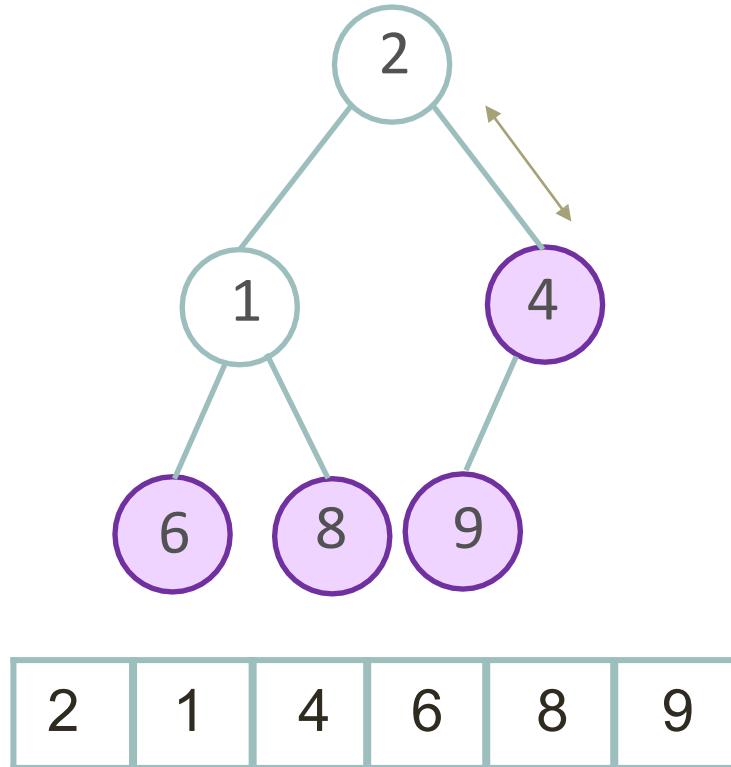
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



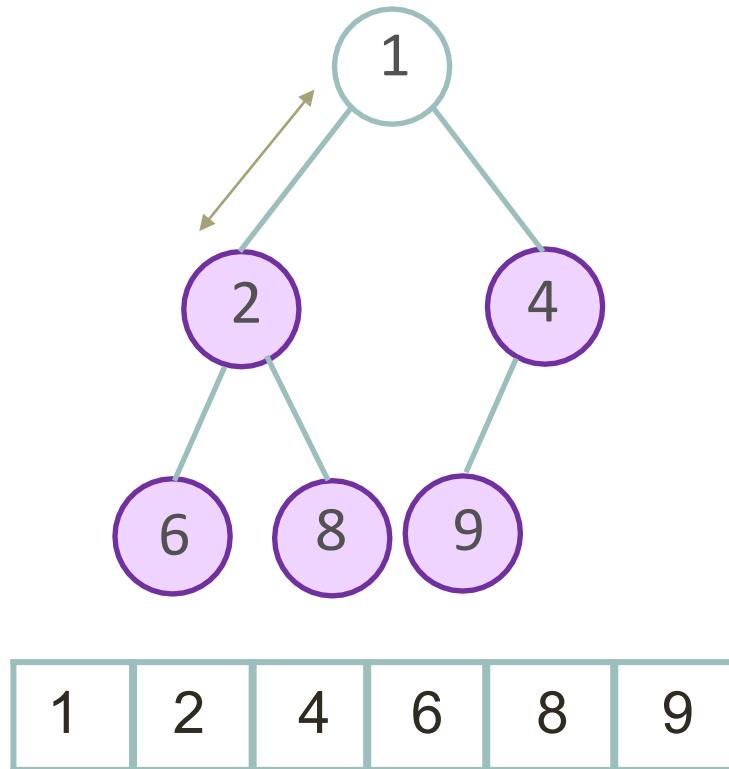
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



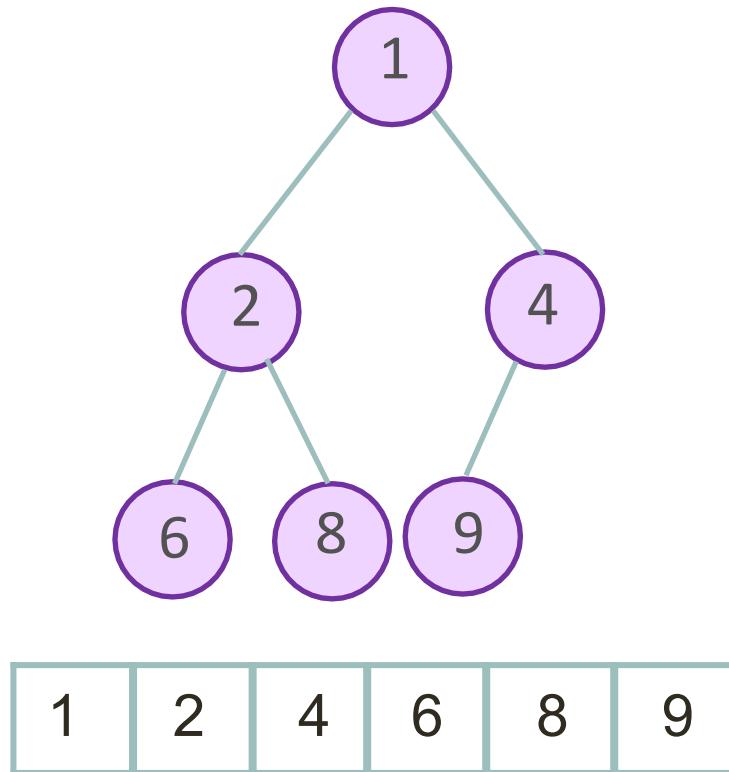
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

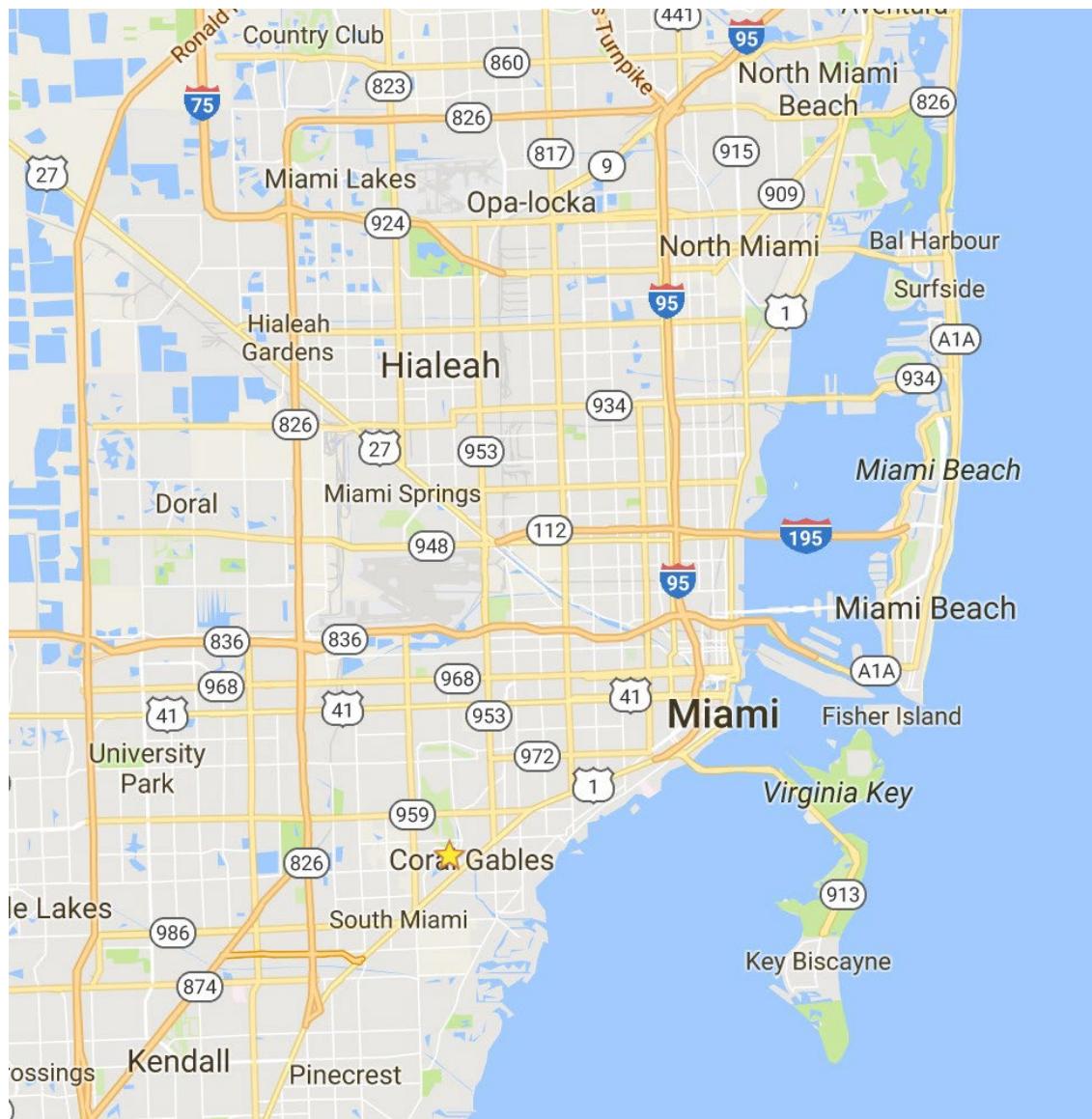
# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

Today...

# Graphs



facebook



Courtesy of Paul Butler

# Flight map



- Graphs
- Paths
- Depth-first search
- Breadth-first search

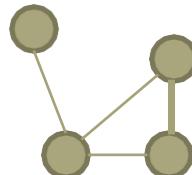
# Graphs

- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional

- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional
- Trees are a *subset* of graphs

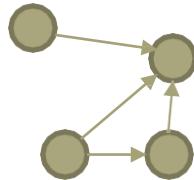
- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional
- Graphs are used **EXTENSIVELY** throughout CS

# Some definitions

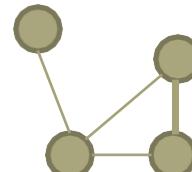


Undirected graph

# Some definitions

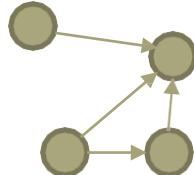


Directed graph

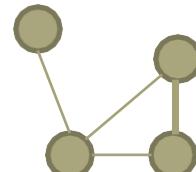


Undirected graph

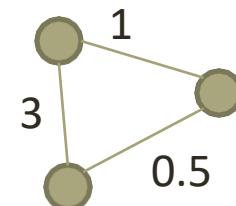
# Some definitions



Directed graph

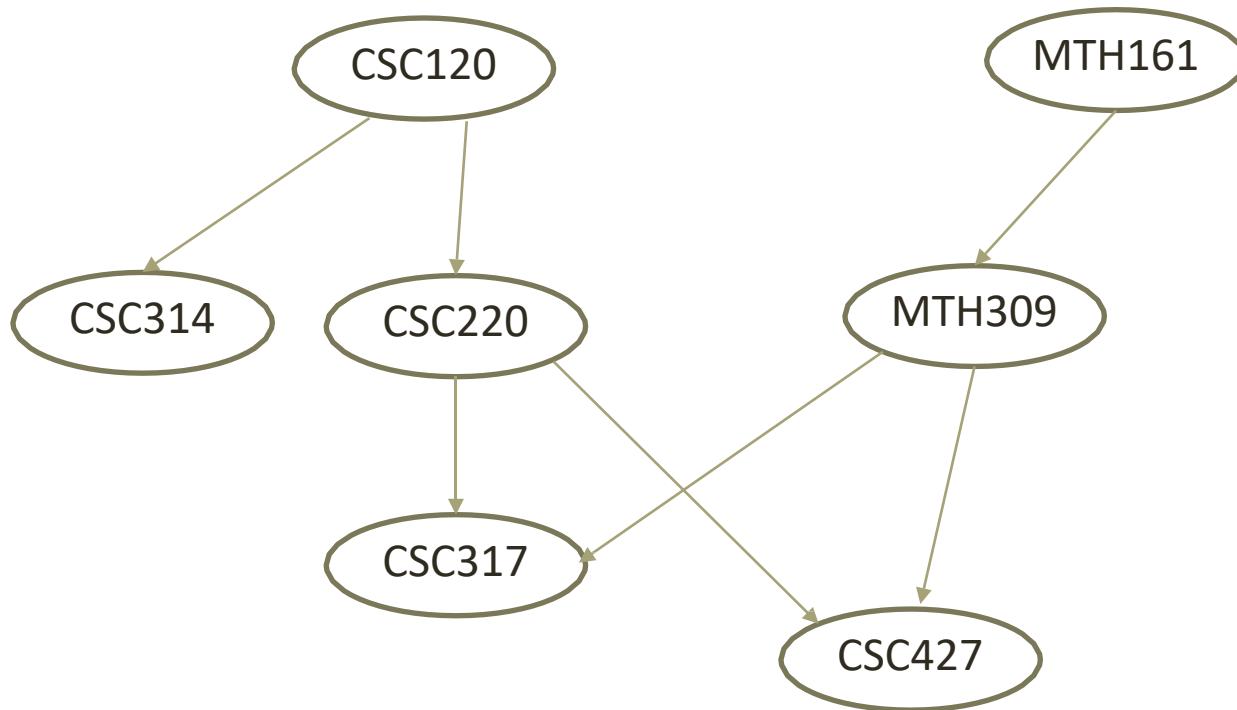


Undirected graph



Weighted graph

# What makes this a graph and not a tree?



- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    . . .  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

- The order in which neighbors appear in the list is unspecified
  - A different order still make the same graph!

# Paths

# Path finding

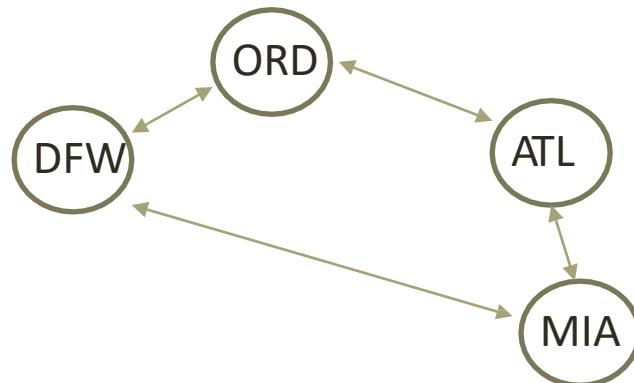
- A **path** is a sequence of nodes with
  - a start-point
  - and an end-point
- ...such that the end-point can be reached through a series of nodes from the start-point

# Path finding

- A **path** is a sequence of nodes with
  - a start-point
  - and an end-point
- ...such that the end-point can be reached through a series of nodes from the start-point

MIA — ATL — ORD

- There is *no direct path*

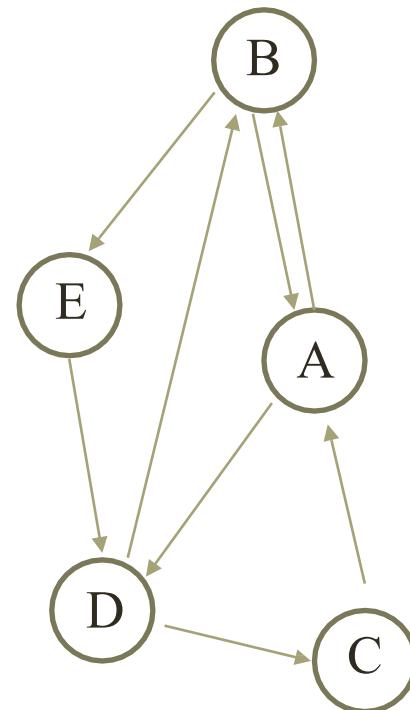


# Cycles

- A cycle in a graph is a path from a node back to itself

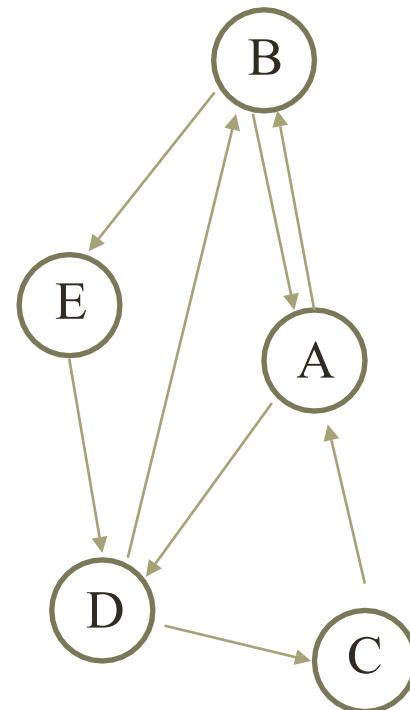
B — E — D — B

- While traversing a graph, special care must be taken to avoid cycles, otherwise what?
- Can trees have cycles?



# Path finding

- There may be more than one path from one node to another
- We are often interested in the *path length*
- Finding the shortest (or cheapest) path between two nodes is a common graph operation

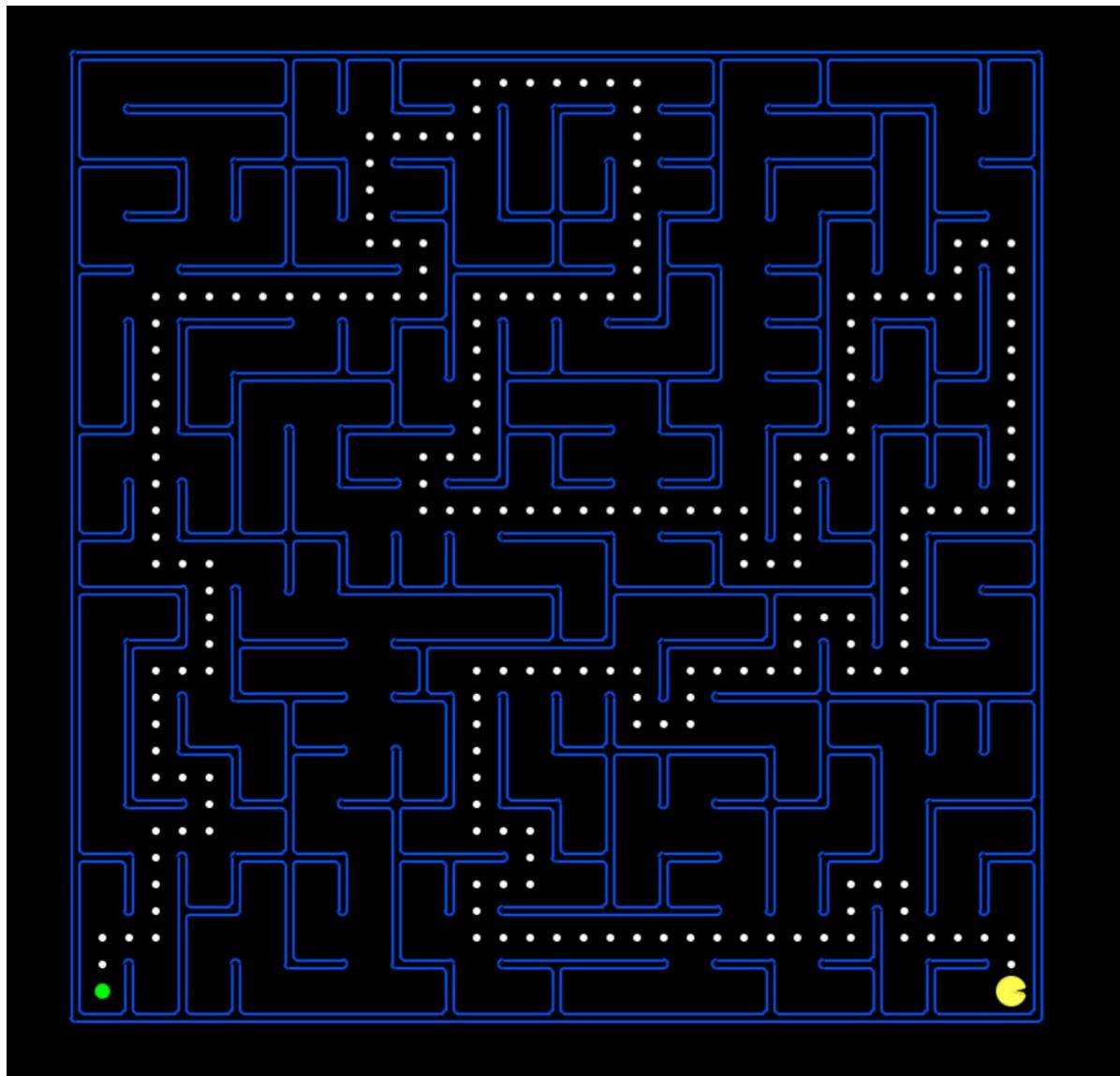


- Any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph

- Any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph
- ....and solved with pathfinding!

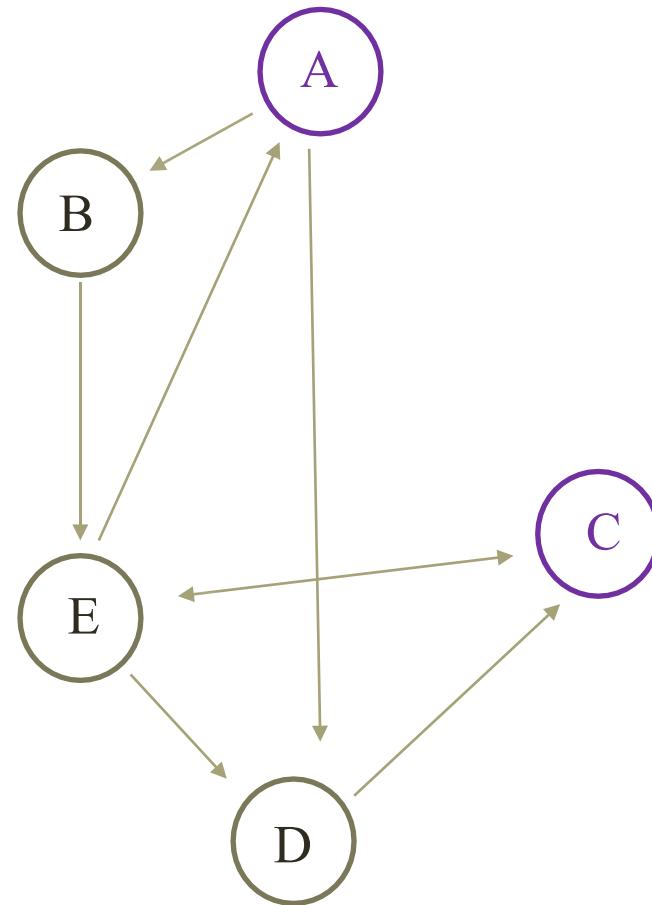
# Example

- In games, moving a character around a space
- Character finds the shortest path from its current location to the destination
  - Not always a straight line!
- Terrain is represented as a graph
  - Every non-obstacle spot on the terrain is a node
  - Nodes are connected to adjacent nodes
  - Navigating a maze...



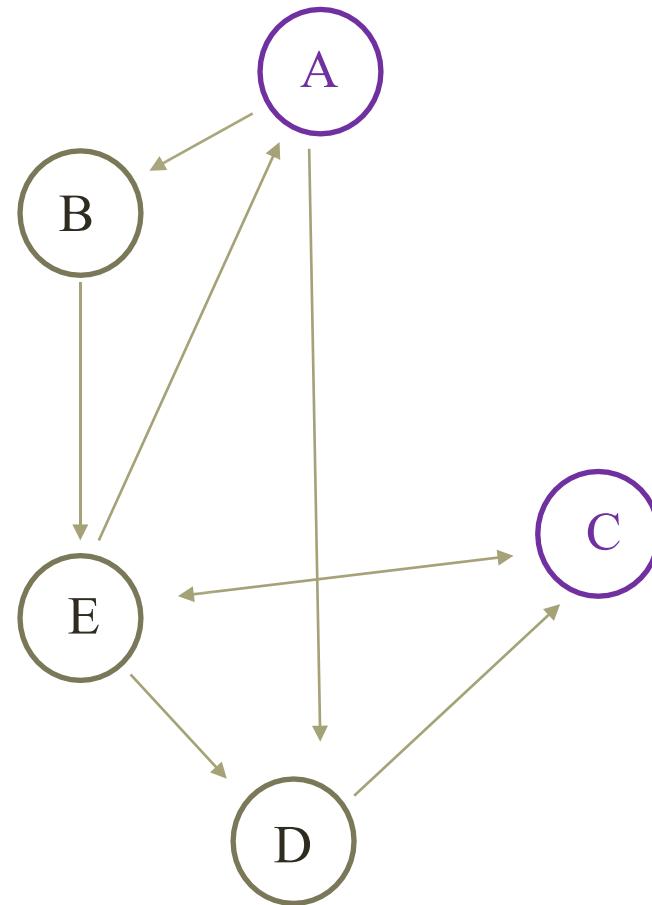
- Depth-first search — DFS
- Breadth-first search — BFS
- If there exists a path from one node to another these algorithms will find it
  - The nodes on this path are the steps to take to get from start point to the end point
- If multiple such paths exist, the algorithms may find different ones

We want to find a path from A to C



# Depth-first search

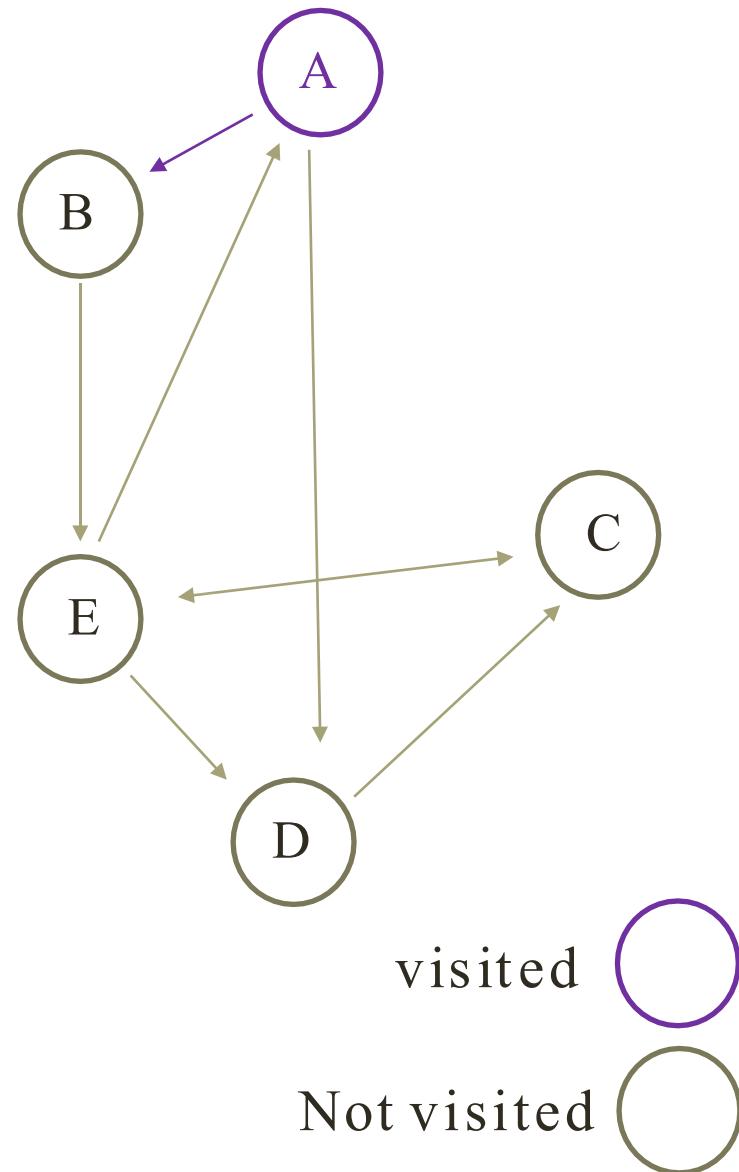
We want to find a path from A to C



We want to find a path from A to C

SO... Start from A,  
traverse its first edge, save  
where we came from, and  
recurse

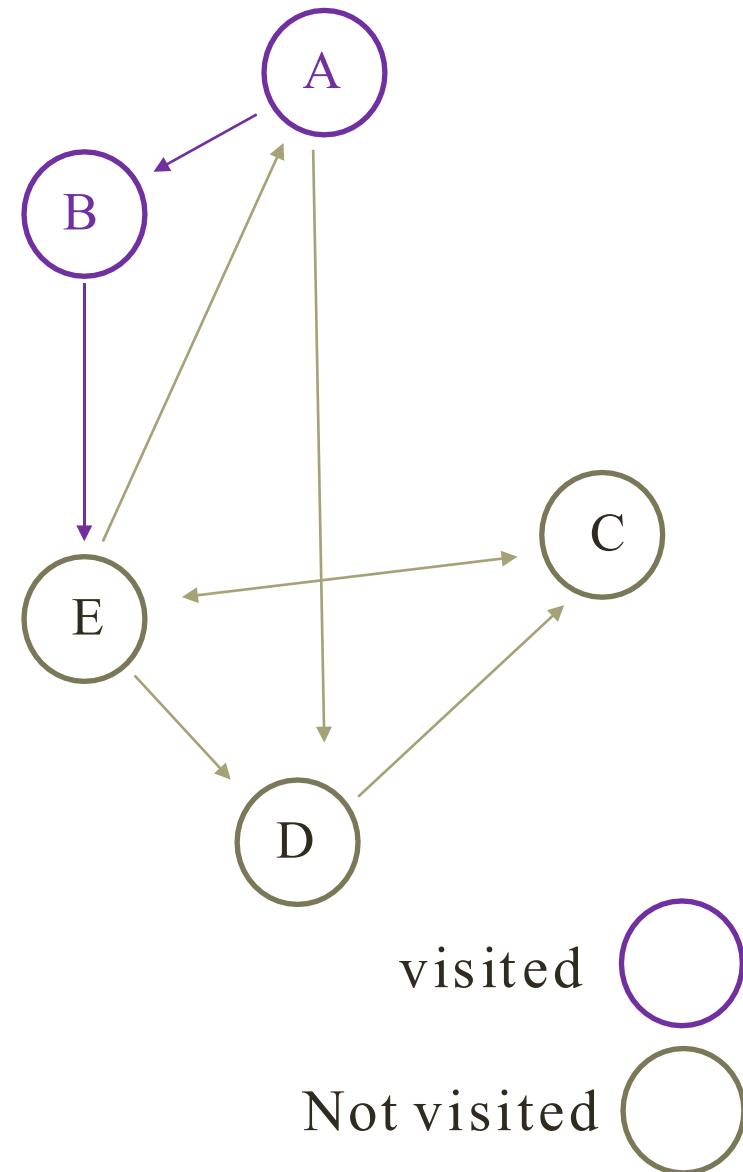
A.visited = true  
B.cameFrom = A



We want to find a path from A to C

Traverse the first unvisited node in the edge list recursively, save where we came from

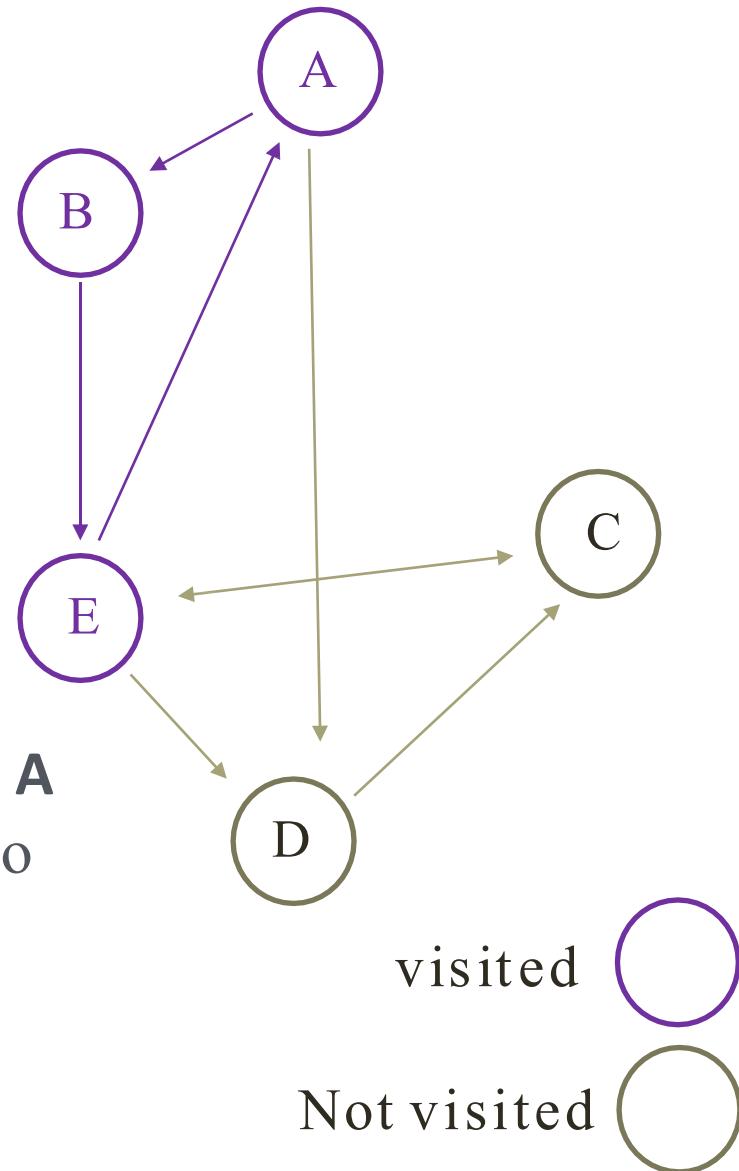
B.visited = true  
E.cameFrom = B



Traverse the first unvisited node in the edge list recursively, save where we came from

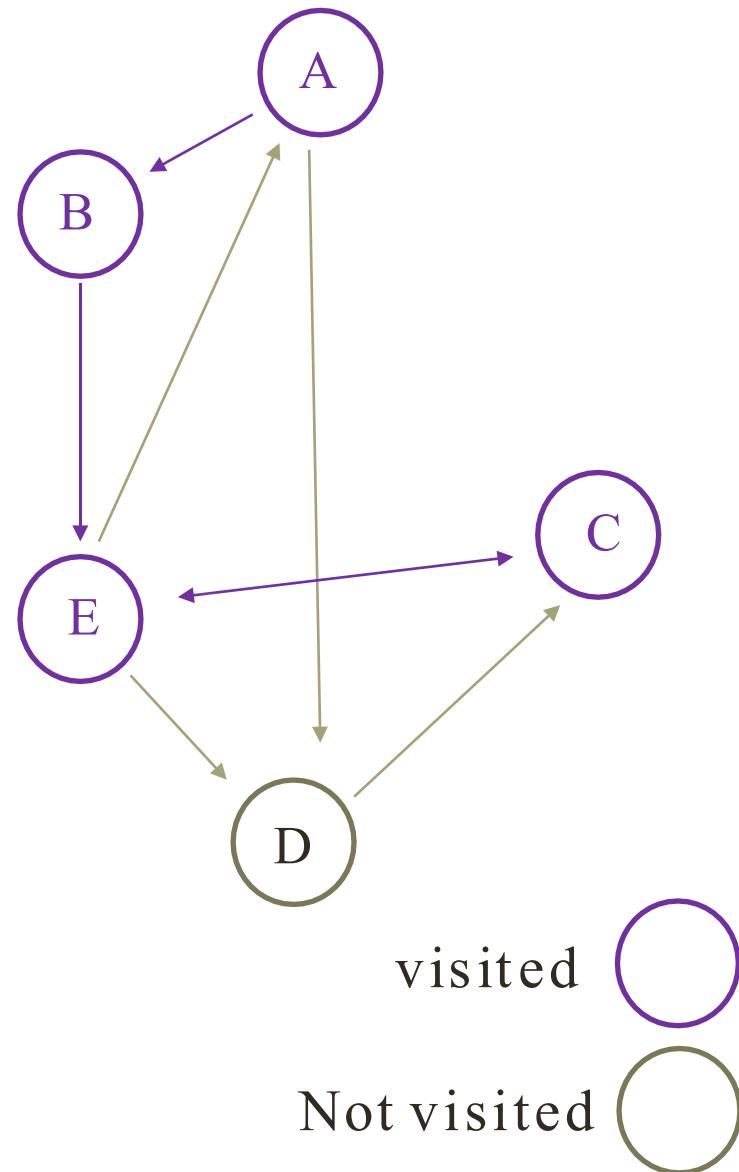
E.visited = true

Look at the first edge; node **A** has already been visited, so skip



Look at next edge; C has not been visited yet

C.cameFrom = E



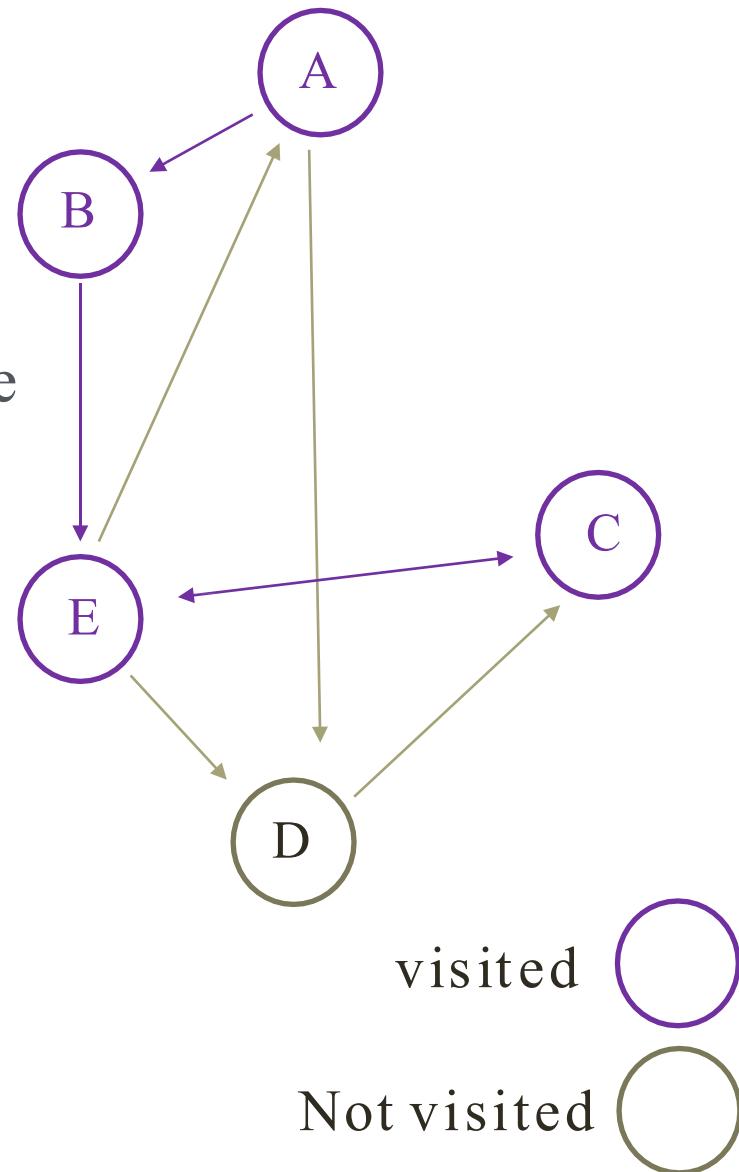
Node C is our goal. we are done!

C.visited = true

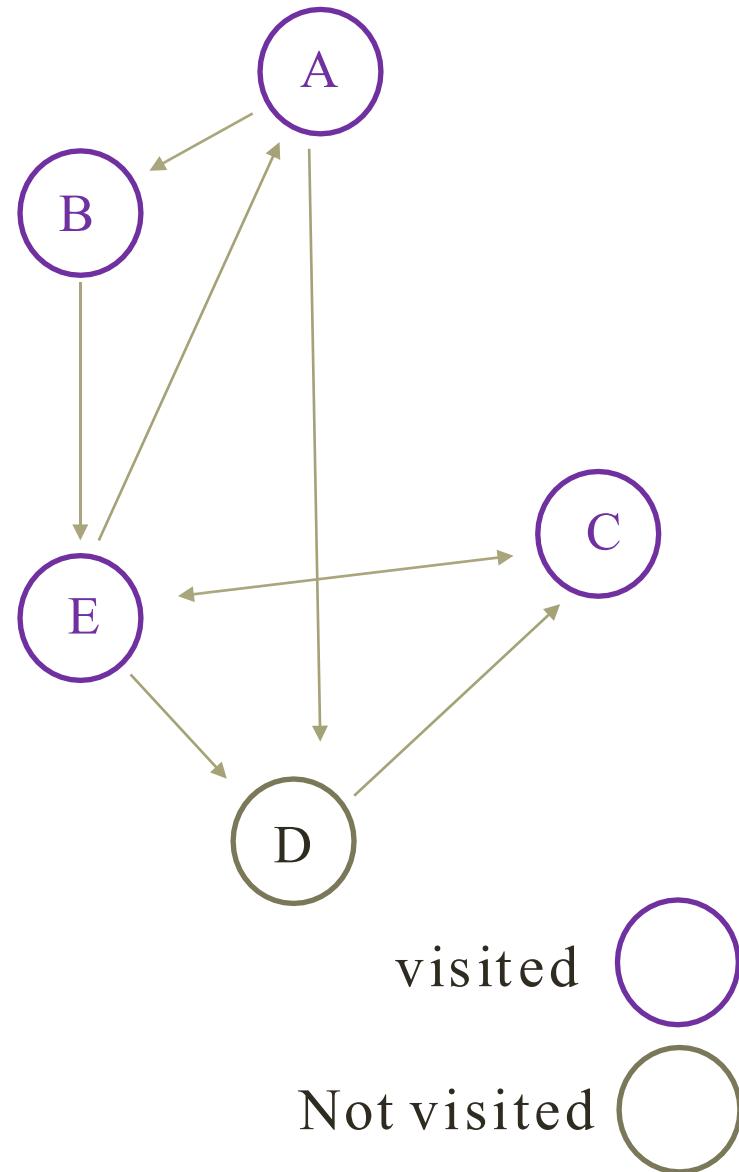
Follow each node's  
**cameFrom** to reconstruct the  
path

C.cameFrom = E,  
E.cameFrom = B,  
B.cameFrom = A

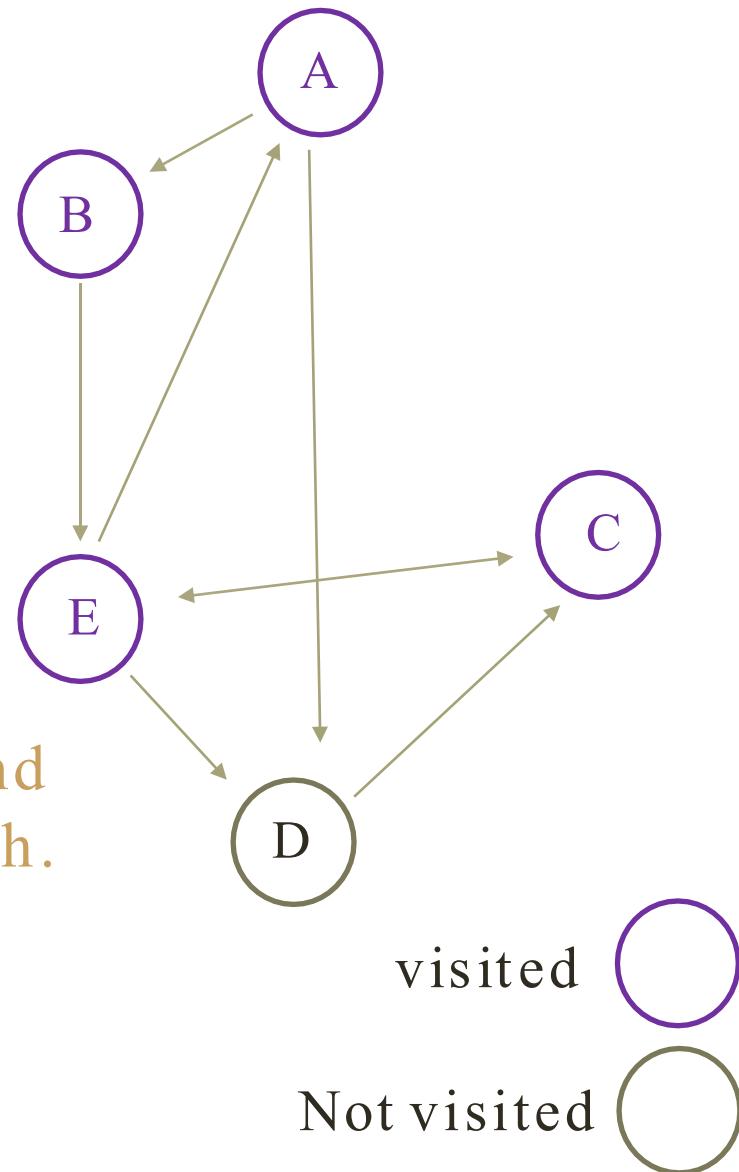
Path: A —B —E —C



Is there a better (shorter) path from A to C?



Is there a better (shorter) path from A to C?



```
DFS(Node curr, Node goal)
{
    curr.visited = true;
    if(curr.equals(goal))
        return;
    for(Node next : curr.neighbors)
        if(!next.visited)
        {
            next.cameFrom = curr;
            DFS(next, goal);
        }
}
// path is now saved in nodes' .cameFrom
```

```
1 // Java program to print DFS traversal from a given given graph
2 import java.io.*;
3 import java.util.*;
4
5 // This class represents a directed graph using adjacency list
6 // representation
7 class Graph
8 {
9     private int v; // No. of vertices
10
11    // Array of lists for Adjacency List Representation
12    private LinkedList<Integer> adj[];
13
14    // Constructor
15    Graph(int v)
16    {
17        this.v = v;
18        adj = new LinkedList[v];
19        for (int i=0; i<v; ++i)
20            adj[i] = new LinkedList();
21    }
22
23    //Function to add an edge into the graph
24    void addEdge(int v, int w)
25    {
26        adj[v].add(w); // Add w to v's list.
27    }
28
29    // A function used by DFS
30    void DFSUtil(int v,boolean visited[])
31    {
32        // Mark the current node as visited and print it
33        visited[v] = true;
34        System.out.print(v+" ");
35
36        // Recur for all the vertices adjacent to this vertex
37        Iterator<Integer> i = adj[v].listIterator();
38        while (i.hasNext())
39        {
40            int n = i.next();
41            if (!visited[n])
42                DFSUtil(n, visited);
43        }
44    }
}
```

```
45
46 // The function to do DFS traversal. It uses recursive DFSUtil()
47 void DFS(int v)
48 {
49     // Mark all the vertices as not visited(set as
50     // false by default in java)
51     boolean visited[] = new boolean[V];
52
53     // Call the recursive helper function to print DFS traversal
54     DFSUtil(v, visited);
55 }
56
57 public static void main(String args[])
58 {
59     Graph g = new Graph(4);
60
61     g.addEdge(0, 1);
62     g.addEdge(0, 2);
63     g.addEdge(1, 2);
64     g.addEdge(2, 0);
65     g.addEdge(2, 3);
66     g.addEdge(3, 3);
67
68     System.out.println("Following is Depth First Traversal "+
69                     "(starting from vertex 2)");
70
71     g.DFS(2);
72 }
73 }
74 // This code is contributed by Aakash Hasija
75
```

# Summary

- Look at the first edge going out of the start node
- Recursively search from the new node
- Upon returning, take the next edge
- If no more edges, return

# Summary

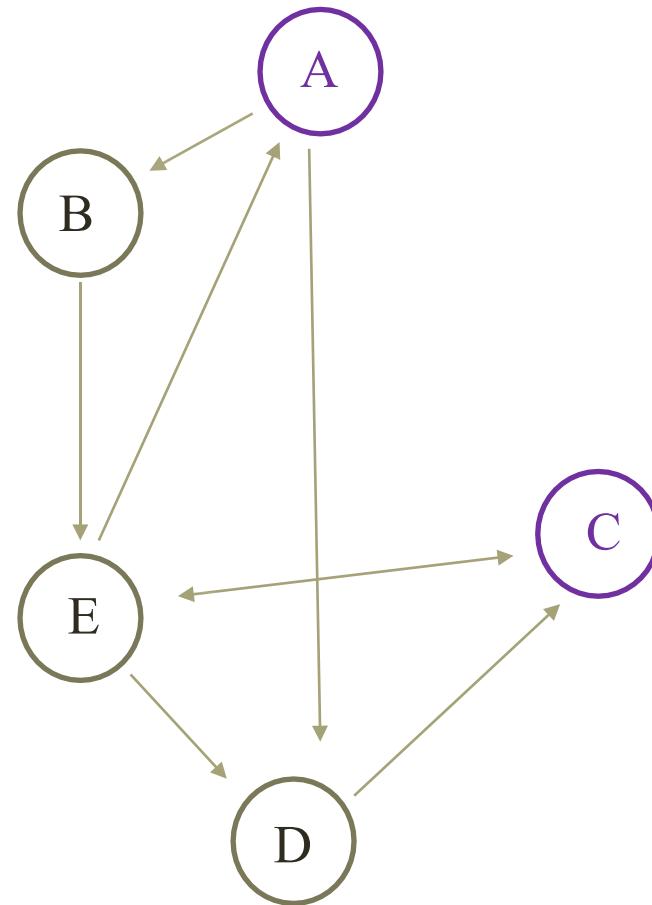
- Look at the first edge going out of the start node
  - Recursively search from the new node
  - Upon returning, take the next edge
  - If no more edges, return
- 
- When visiting a node, mark it as visited
    - So we don't get stuck in a cycle
    - Skip already visited nodes during traversal

# Summary

- Look at the first edge going out of the start node
  - Recursively search from the new node
  - Upon returning, take the next edge
  - If no more edges, return
- 
- When visiting a node, mark it as visited
    - So we don't get stuck in a cycle
    - Skip already visited nodes during traversal
  - For each node visited, save a reference to the node where we came from to reconstruct the path

# Breadth-first search

We want to find a path from A to C

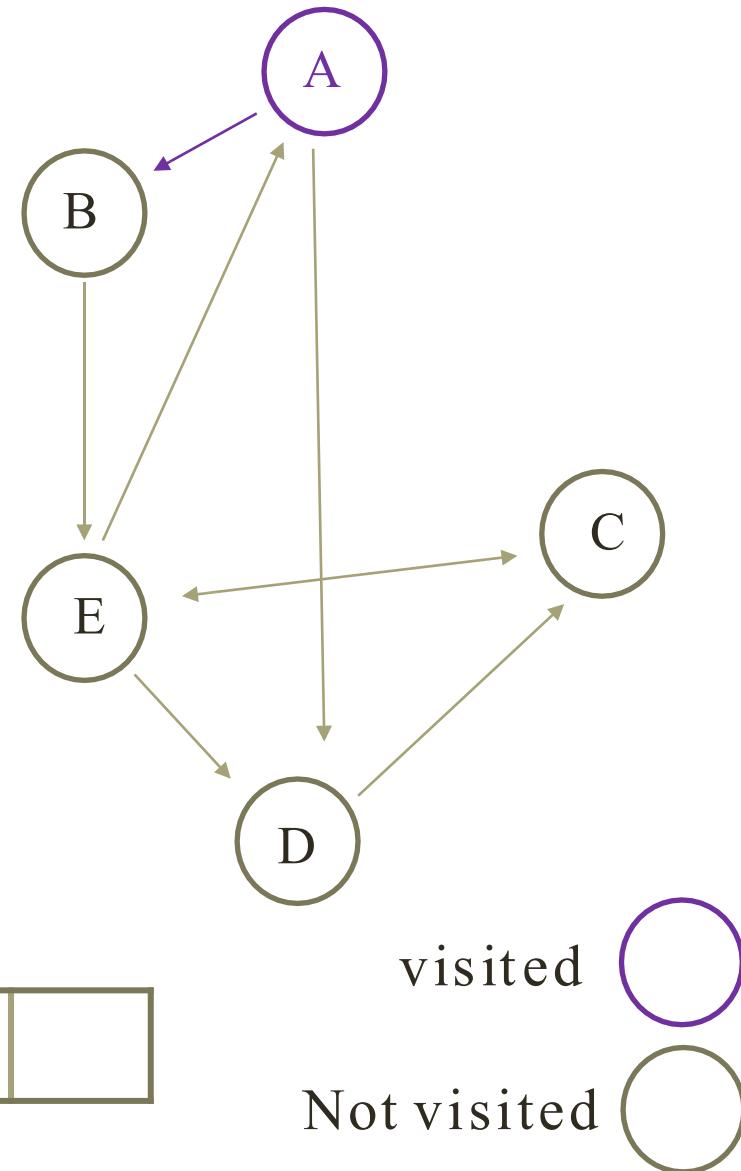


We want to find a path from A to C

Mark and enqueue the start node A

A.visited = true

Queue

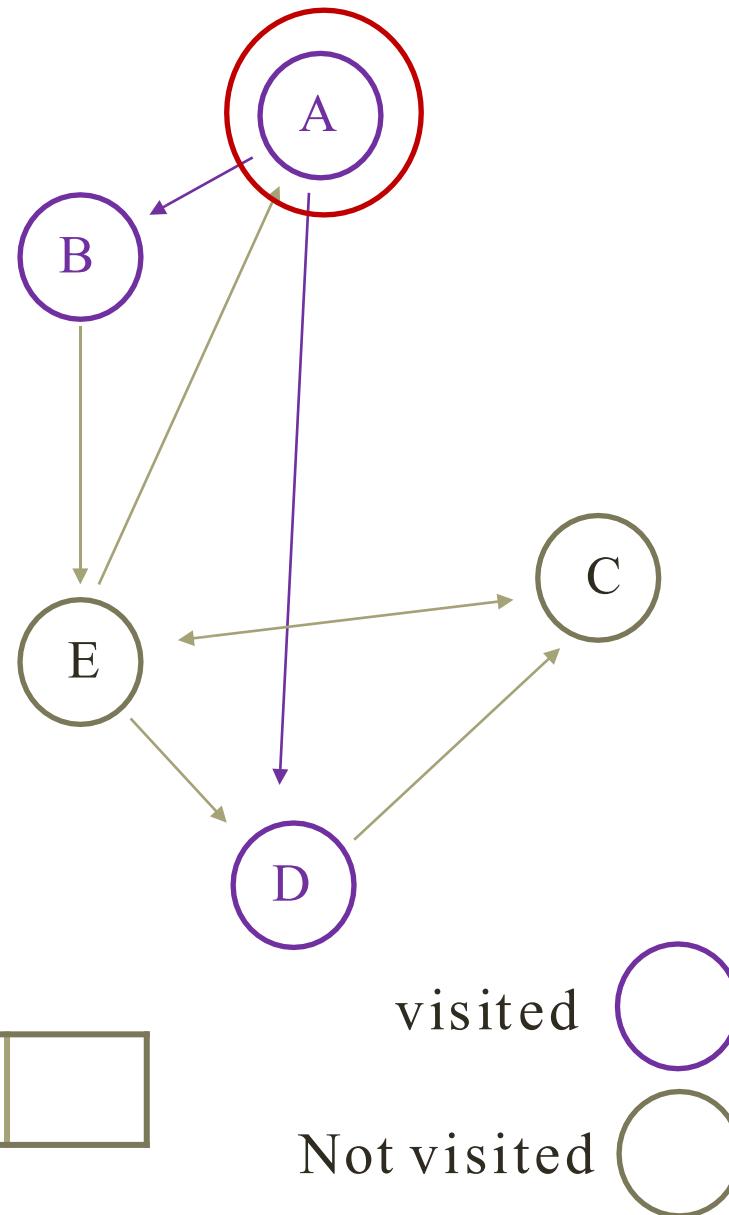


Dequeue the first node in the queue (A)

Mark and enqueue A's unvisited neighbors

```
B.cameFrom = A  
D.cameFrom = A  
B.visited = true  
D.visited = true
```

Queue [ B | D | ]

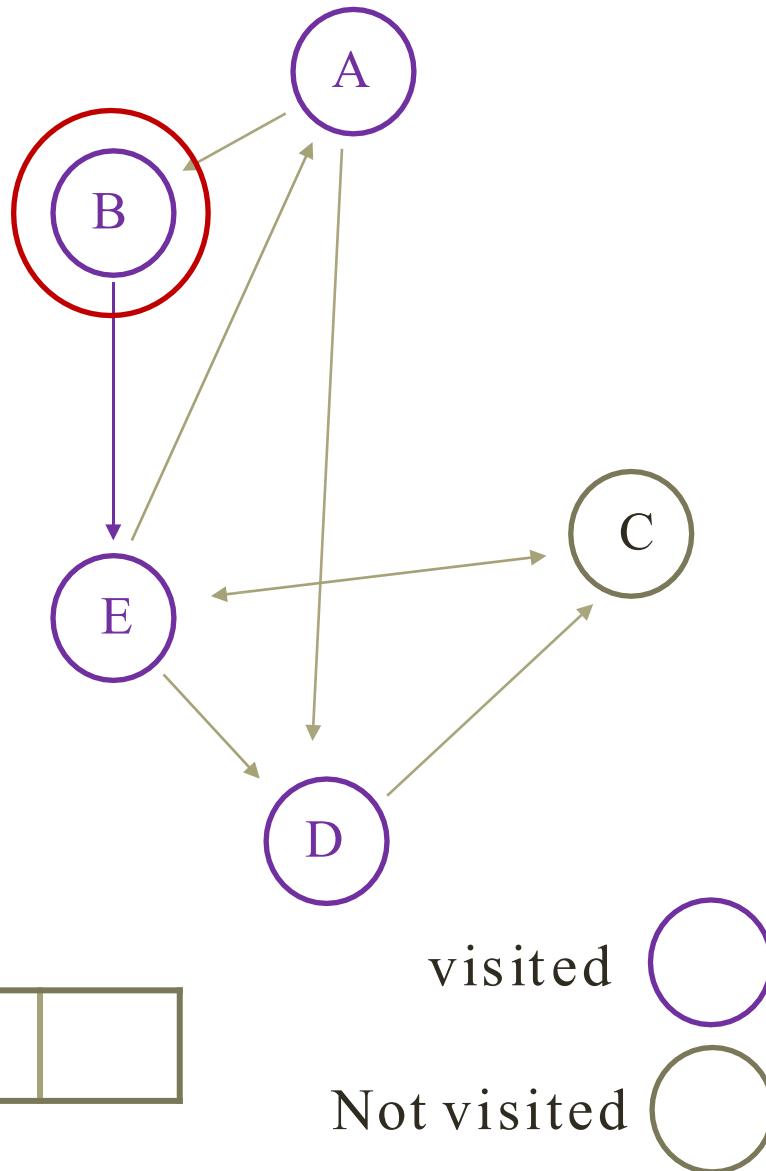


Dequeue the first node in the queue (B)

Mark and enqueue B's unvisited neighbors

E.cameFrom = B  
E.visited = true

Queue [D | E | ]

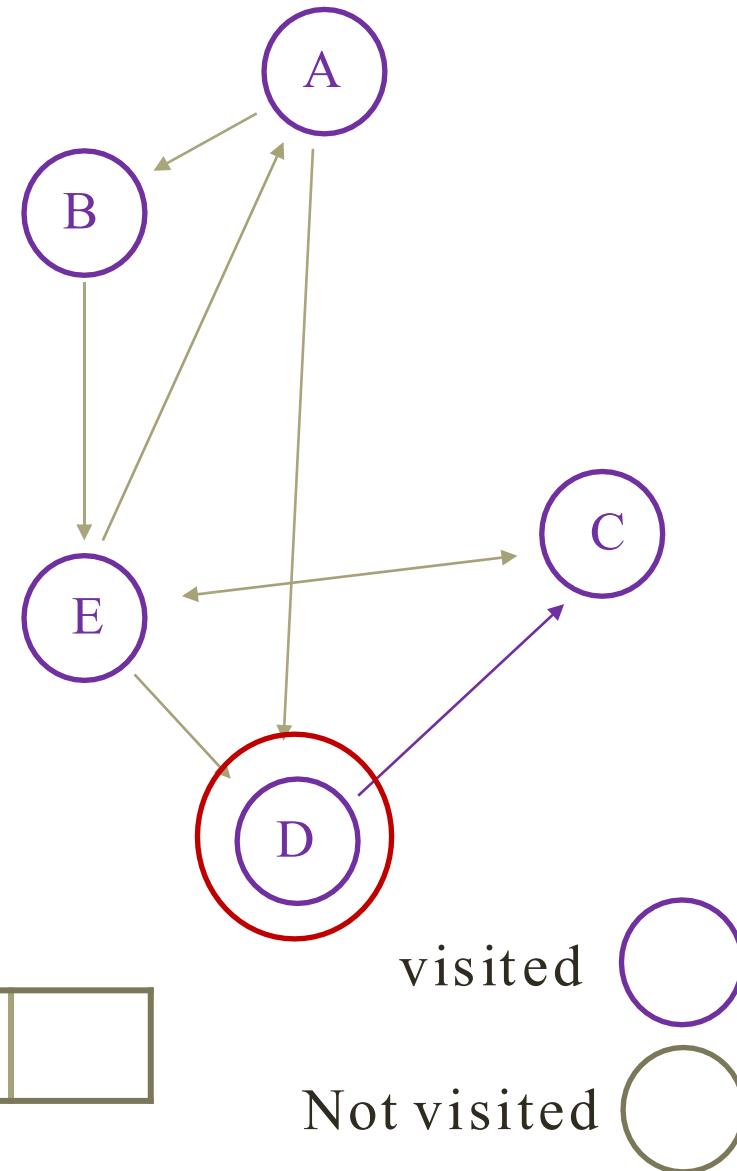


Dequeue the first node in the queue (D)

Mark and enqueue D's  
unvisited neighbors

C.cameFrom = D  
C.visited = true

Queue [E | C | ]

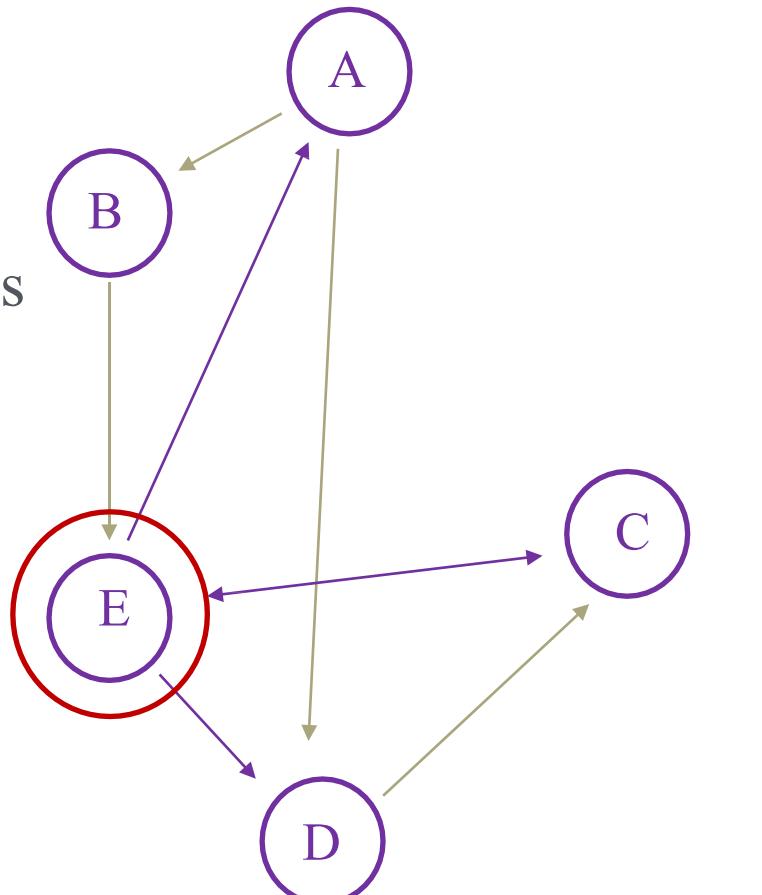


Dequeue the first node in the queue (E)

Mark and enqueue E's  
unvisited visited neighbors

(no unvisited neighbors!)

Queue



visited

Not visited

Dequeue the first node in the queue (C)

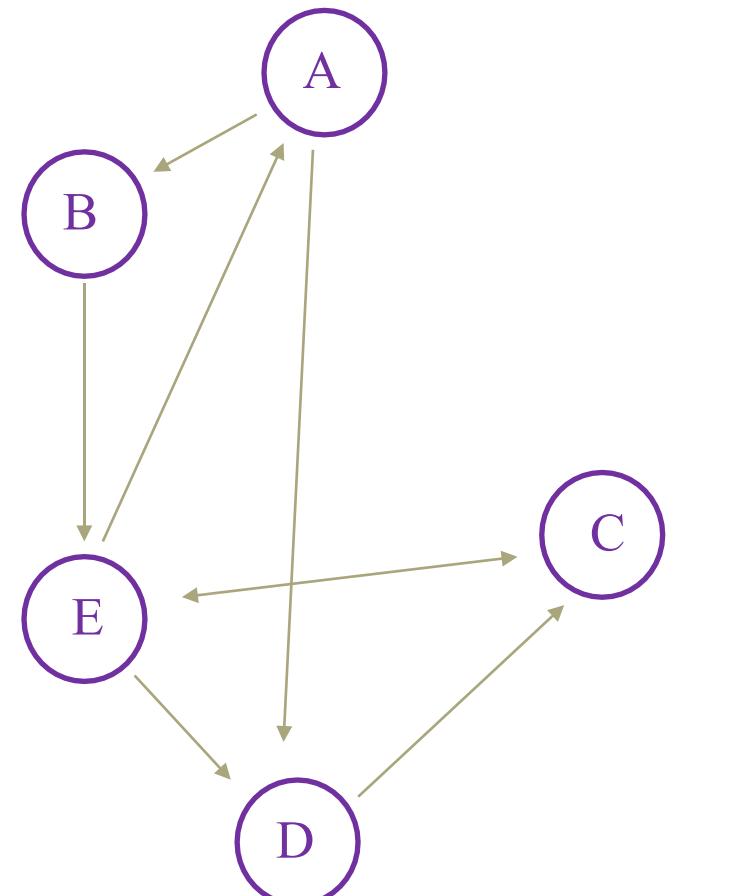
C is the goal! Reconstruct  
the path visited  
with **cameFrom** references

C.cameFrom = D

D.cameFrom = A

Path: A —D —C

Queue

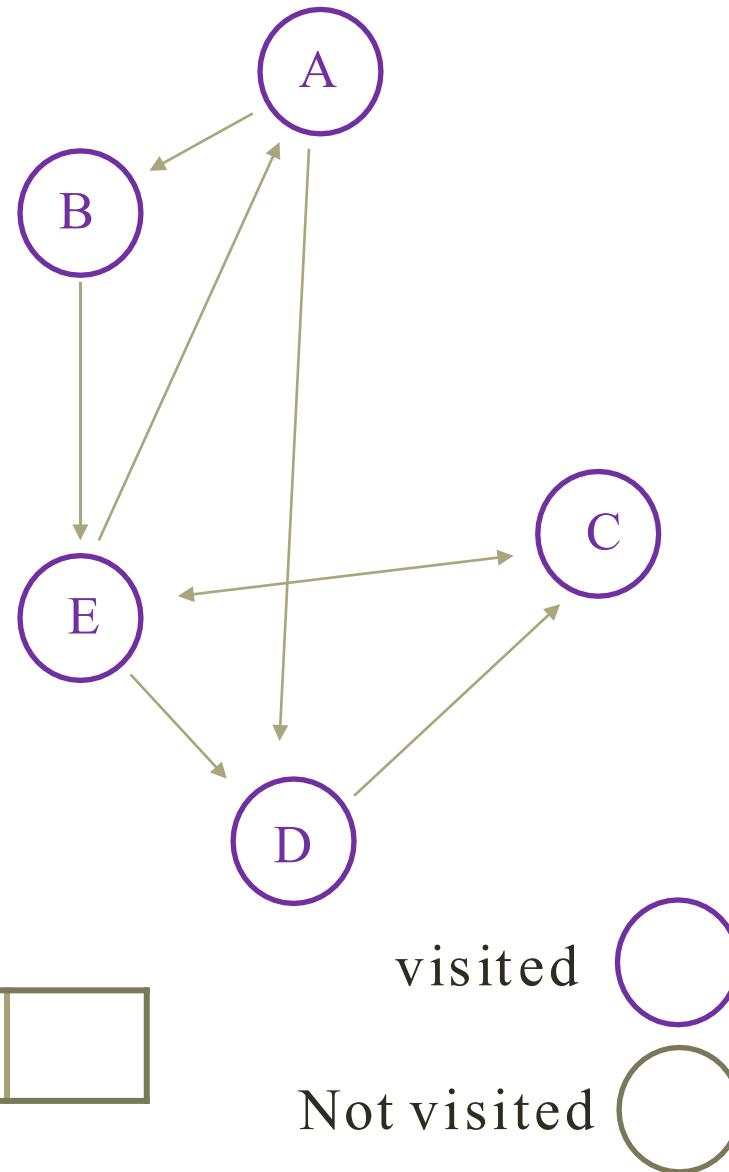


visited

Not visited

Is this the shortest path?

Path: A — D — C



Queue



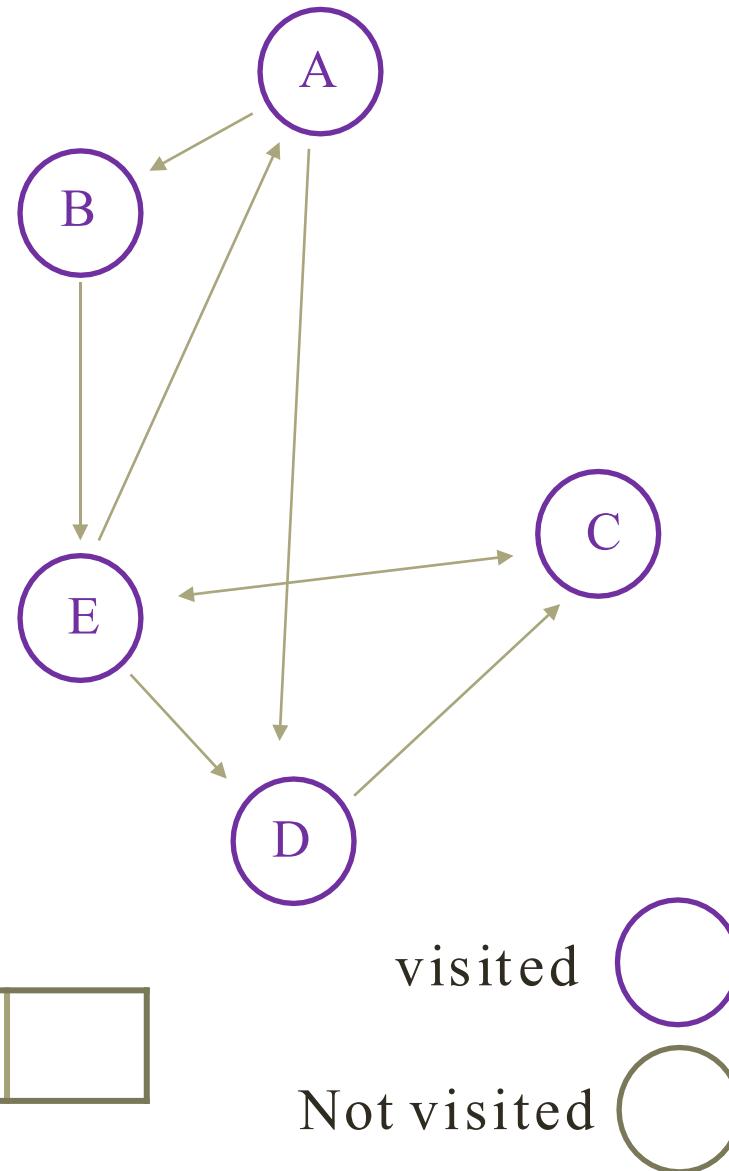
Is this the shortest path?

Path: A — D — C

BFS visits nodes closest to the start-point first

Therefore, the first path found is the shortest path (closest to the start node)

Queue



```
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)
    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return
        for(Node next : curr.neighbors)
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
```

```
1 // Java program to print BFS traversal from a given source vertex.
2 // BFS(int s) traverses vertices reachable from s.
3 import java.io.*;
4 import java.util.*;
5
6 // This class represents a directed graph using adjacency list
7 // representation
8 class Graph
9 {
10     private int v; // No. of vertices
11     private LinkedList<Integer> adj[]; //Adjacency Lists
12
13     // Constructor
14     Graph(int v)
15     {
16         V = v;
17         adj = new LinkedList[v];
18         for (int i=0; i<v; ++i)
19             adj[i] = new LinkedList();
20     }
21
22     // Function to add an edge into the graph
23     void addEdge(int v,int w)
24     {
25         adj[v].add(w);
26     }
```

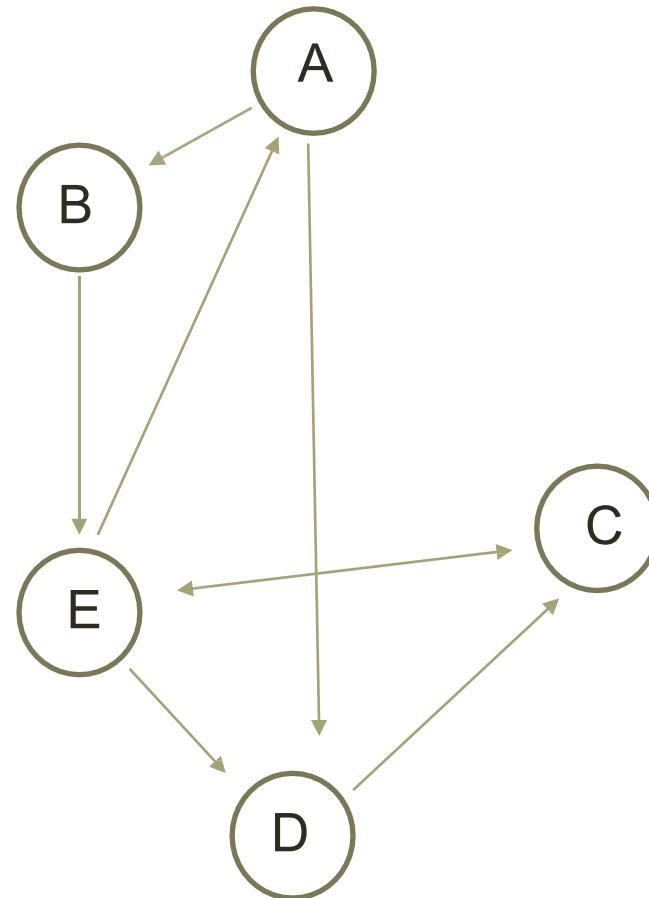
```
27
28 // prints BFS traversal from a given source s
29 void BFS(int s)
30 {
31     // Mark all the vertices as not visited(By default
32     // set as false)
33     boolean visited[] = new boolean[V];
34
35     // Create a queue for BFS
36     LinkedList<Integer> queue = new LinkedList<Integer>();
37
38     // Mark the current node as visited and enqueue it
39     visited[s]=true;
40     queue.add(s);
41
42     while (queue.size() != 0)
43     {
44         // Dequeue a vertex from queue and print it
45         s = queue.poll();
46         System.out.print(s+" ");
47
48         // Get all adjacent vertices of the dequeued vertex s
49         // If a adjacent has not been visited, then mark it
50         // visited and enqueue it
51         Iterator<Integer> i = adj[s].listIterator();
52         while (i.hasNext())
53         {
54             int n = i.next();
55             if (!visited[n])
56             {
57                 visited[n] = true;
58                 queue.add(n);
59             }
60         }
61     }
62 }
```

```
64 // Driver method to
65 public static void main(String args[])
66 {
67     Graph g = new Graph(4);
68
69     g.addEdge(0, 1);
70     g.addEdge(0, 2);
71     g.addEdge(1, 2);
72     g.addEdge(2, 0);
73     g.addEdge(2, 3);
74     g.addEdge(3, 3);
75
76     System.out.println("Following is Breadth First Traversal "+
77                         "(starting from vertex 2)");
78
79     g.BFS(2);
80 }
81 }
82 // This code is contributed by Aakash Hasija
83 }
```

- Instead of visiting deeper nodes first, visit shallower nodes first
  - Visit nodes closest to the start point first, gradually get further away
- Create an empty queue
- Put the starting node in the queue
- While the queue is not empty
  - Dequeue the current node
  - For each unvisited neighbor of the current node
    - *Mark the neighbor as visited*
    - *Put the neighbor into the queue*
- Notice it is not recursive... it just runs until the queue is empty!

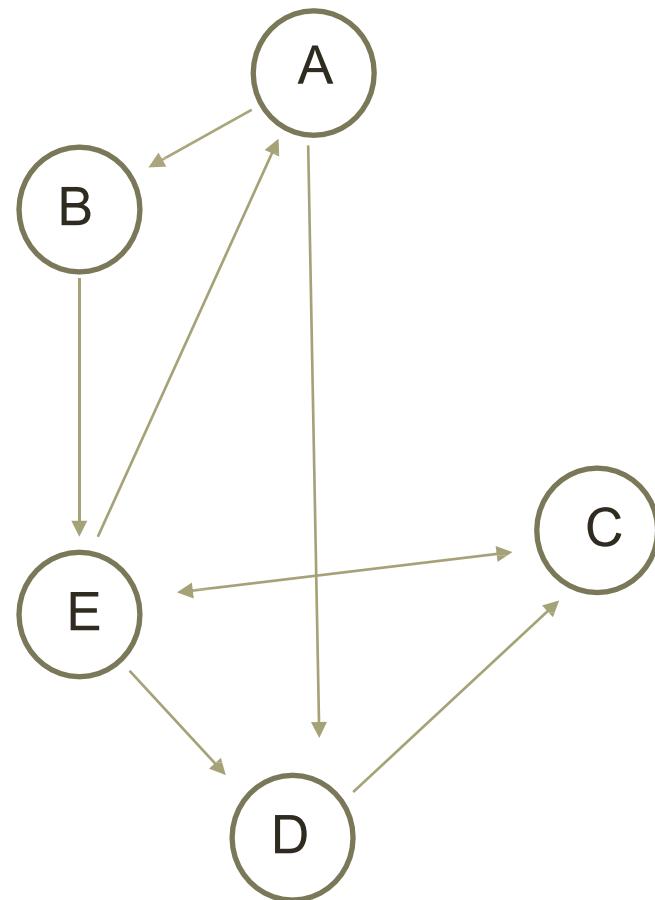
What path will BFS find from B to C?

- A) B E C
- B) B E A D C
- C) B E D C
- D) none



What path will DFS find from A to D?

- A) A B E D
- B) A D
- C) none
- D) this is a trick question



What is true of DFS, searching from a start node to a goal node?

- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a and b
- E) a and c
- F) a, b, and c

What is true of BFS, searching from a start node to a goal node?

- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a, and b
- F) a, and c

# Summary

- Depth-first search — DFS
- Breadth-first search — BFS
- If there exists a path from one node to another these algorithms will find it
  - The nodes on this path are the steps to take to get from point A to point B
- If multiple such paths exist, the algorithms may find different ones

Next time...

- Continue with graphs
- Quiz
- Start on your assignment early!

# Graphs

CSC220|Computer Programming 2

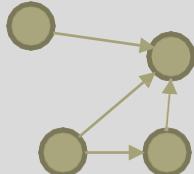
# Administrivia...

- Start your assignment early
  - Involves a lot of details
  - Counts twice as much!
- Start preparing for final

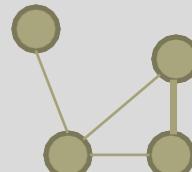
Last Time...

- A **graph** is a set of **nodes** connected by **edges**
  - An edge is just a link between two nodes
  - Nodes don't have a parent-child relationship
  - Links can be bi-directional
- Trees are a *subset* of graphs

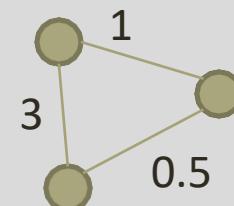
# Some definitions



Directed graph

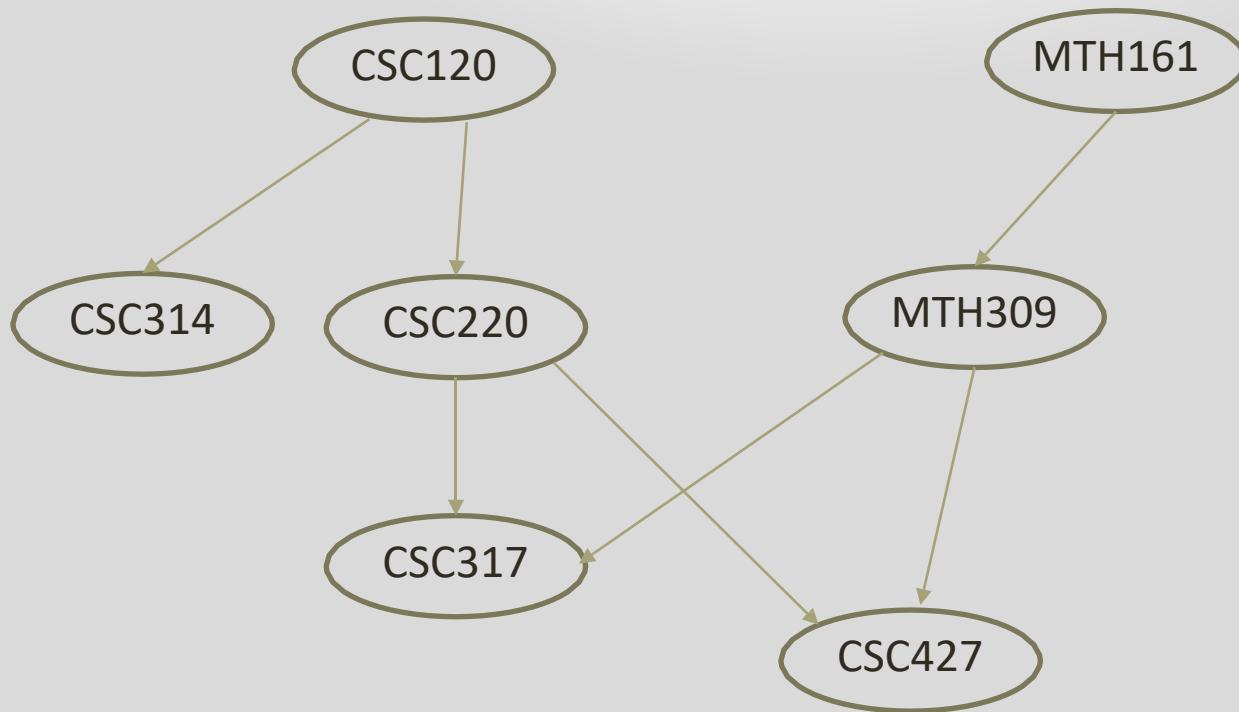


Undirected graph



Weighted graph

# What makes this a graph and not a tree?



- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    . . .  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

- Graphs have no root; must store all nodes

```
class Graph<E> {  
    List<Node> nodes;  
    ...  
}
```

- Implementation is more general than a tree

```
class Node {  
    E Data;  
    List<Node> neighbors;  
    ...  
}
```

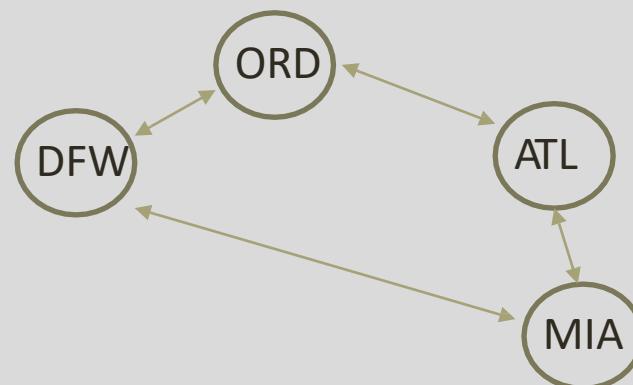
- The order in which neighbors appear in the list is unspecified
  - A different order still make the same graph!

# Path finding

- A **path** is a sequence of nodes with
  - a start-point
  - and an end-point
- ...such that the end-point can be reached through a series of nodes from the start-point

MIA — ATL — ORD

- There is *no direct path*

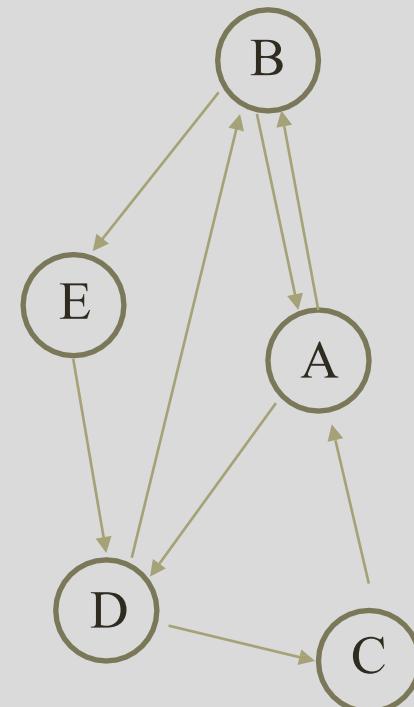


# Cycles

- A cycle in a graph is a path from a node back to itself

B — E — D — B

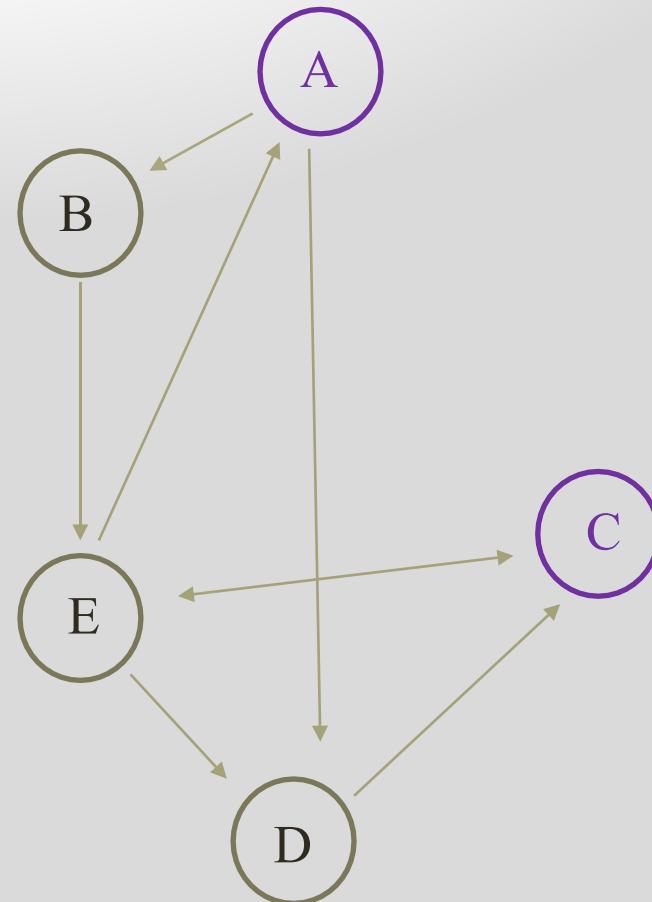
- While traversing a graph, special care must be taken to avoid cycles, otherwise what?
- Can trees have cycles?



- Depth-first search — DFS
  - Not guaranteed to find the shortest path, just a path.
- Breadth-first search — BFS
  - The first path found is the shortest path (closest to the start node)

# Depth-first search

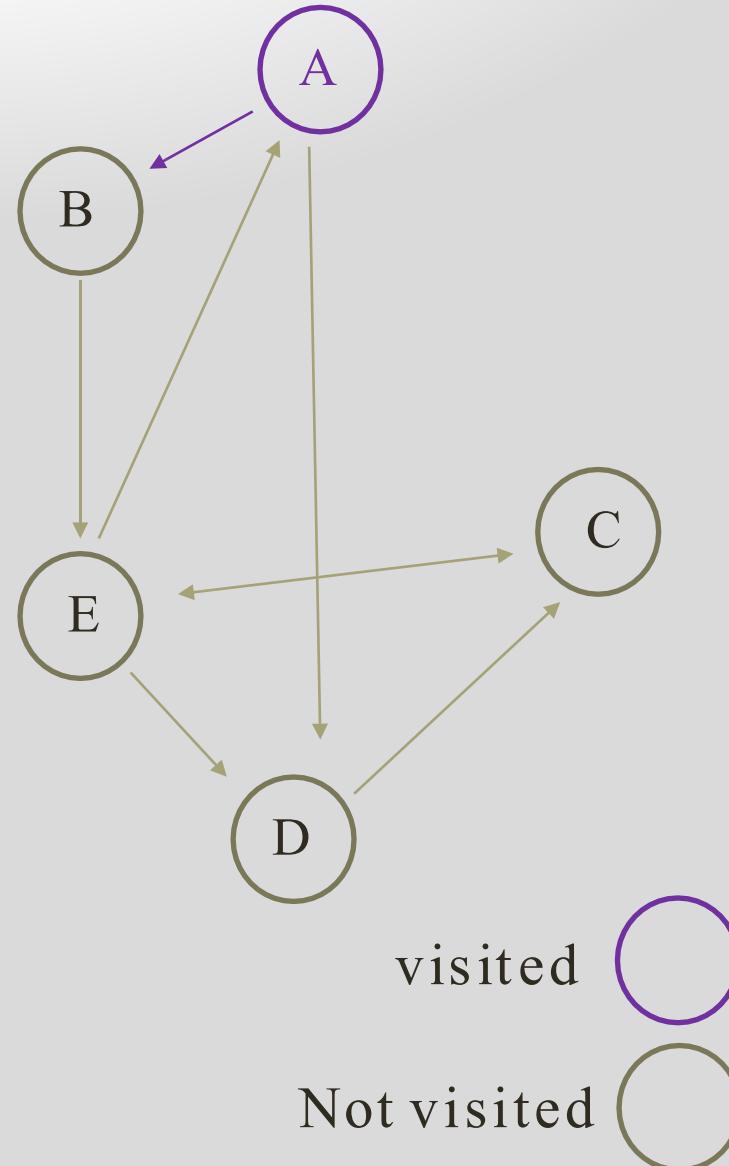
We want to find a path from A to C



We want to find a path from A to C

Start from **A**, traverse its first edge, save where we came from, and recurse

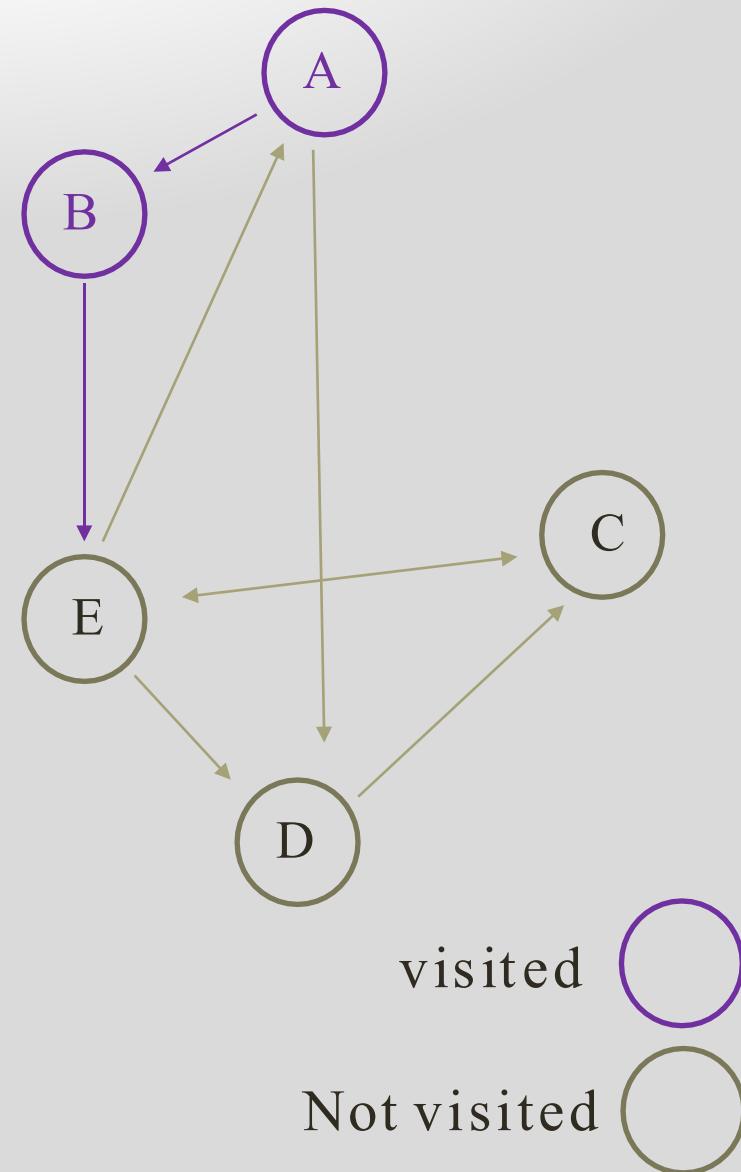
A.visited = true  
B.cameFrom = A



We want to find a path from A to C

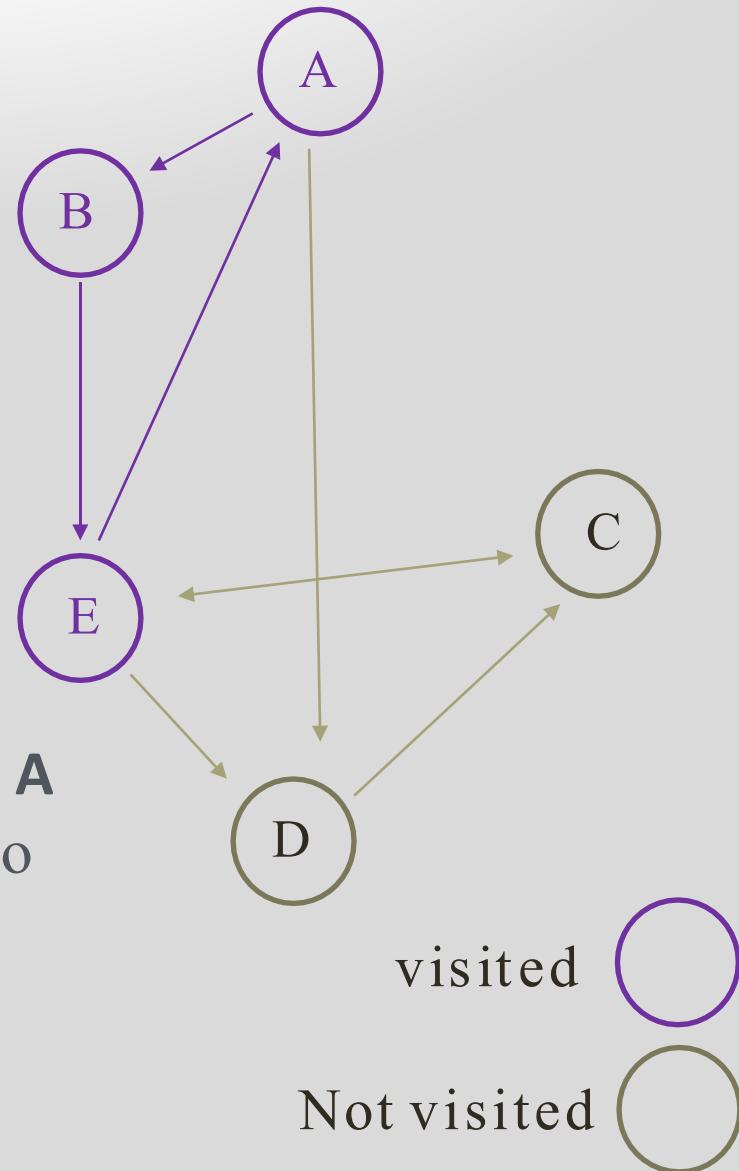
Traverse the first unvisited node in the edge list recursively, save where we came from

B.visited = true  
E.cameFrom = B



Traverse the first unvisited node in the edge list recursively, save where we came from

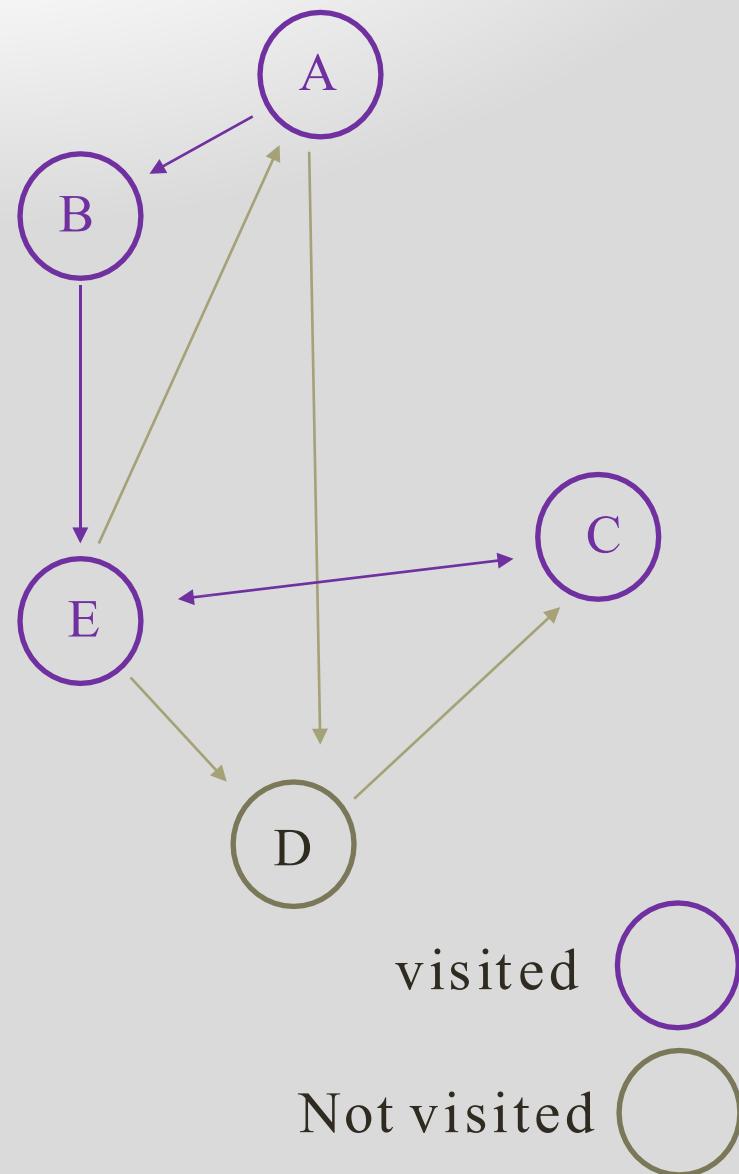
E.visited = true



Look at the first edge; node **A** has already been visited, so skip

Look at next edge; C has not been visited yet

C.cameFrom = E



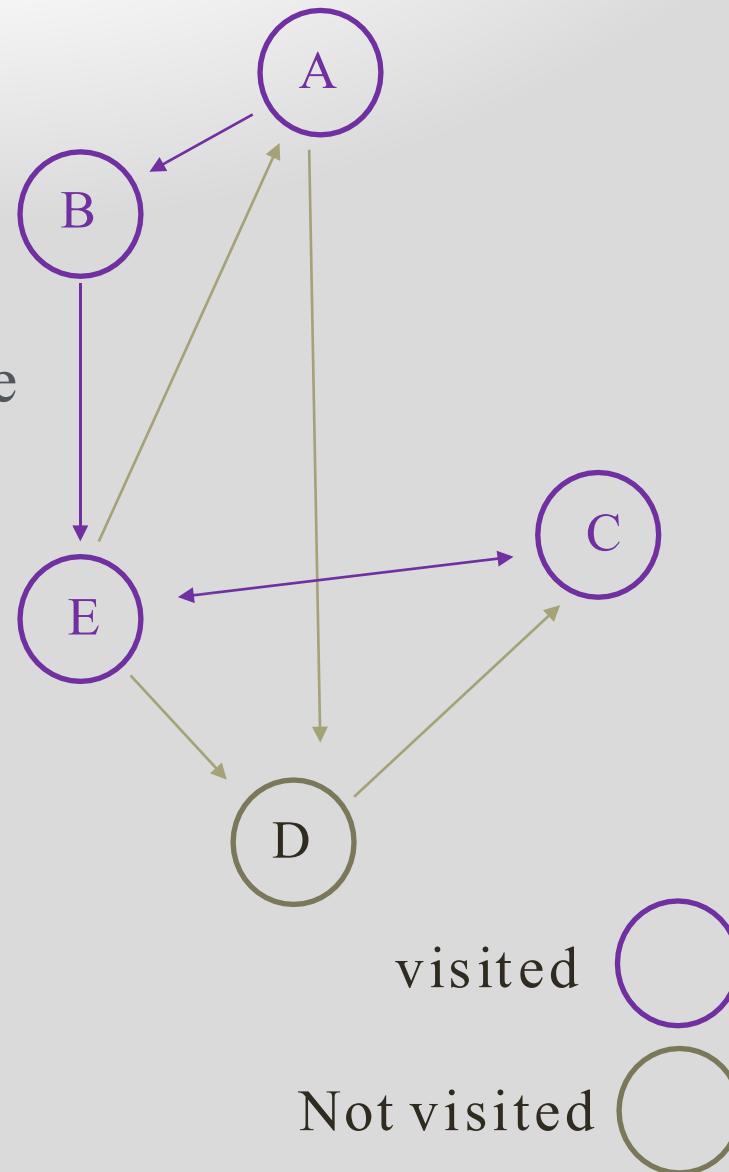
Node C is our goal. we are done!

C.visited = true

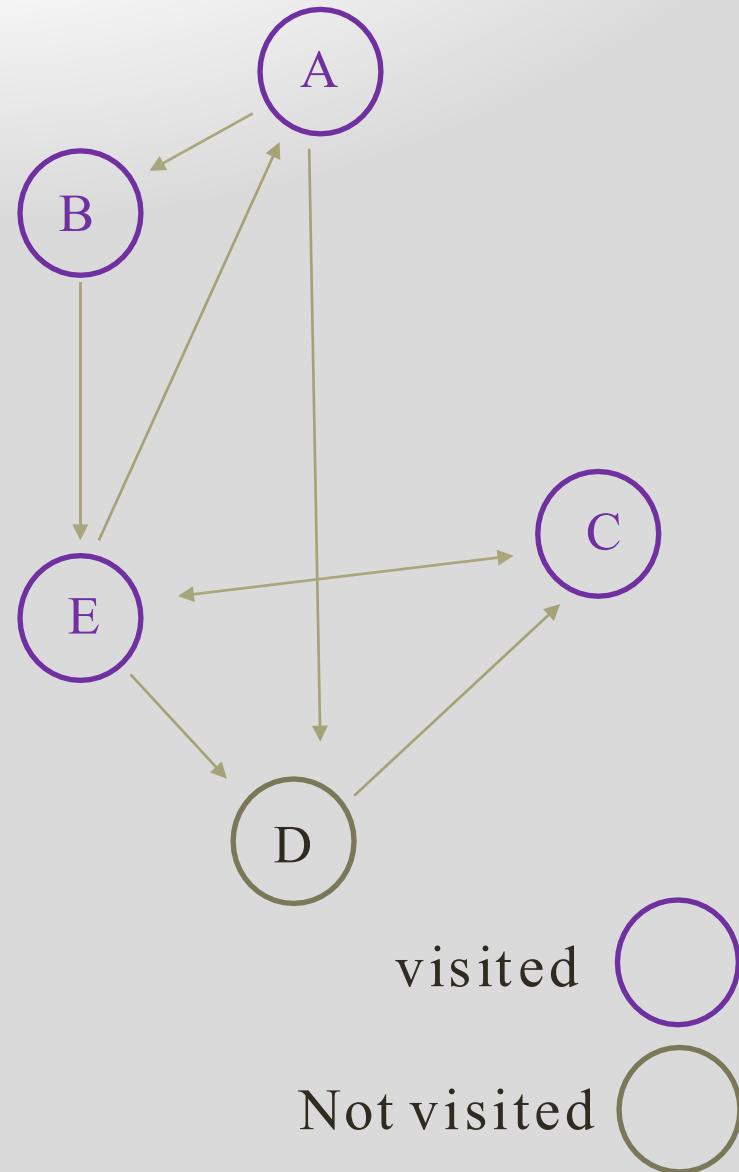
Follow each node's **cameFrom** to reconstruct the path

C.cameFrom = E,  
E.cameFrom = B,  
B.cameFrom = A

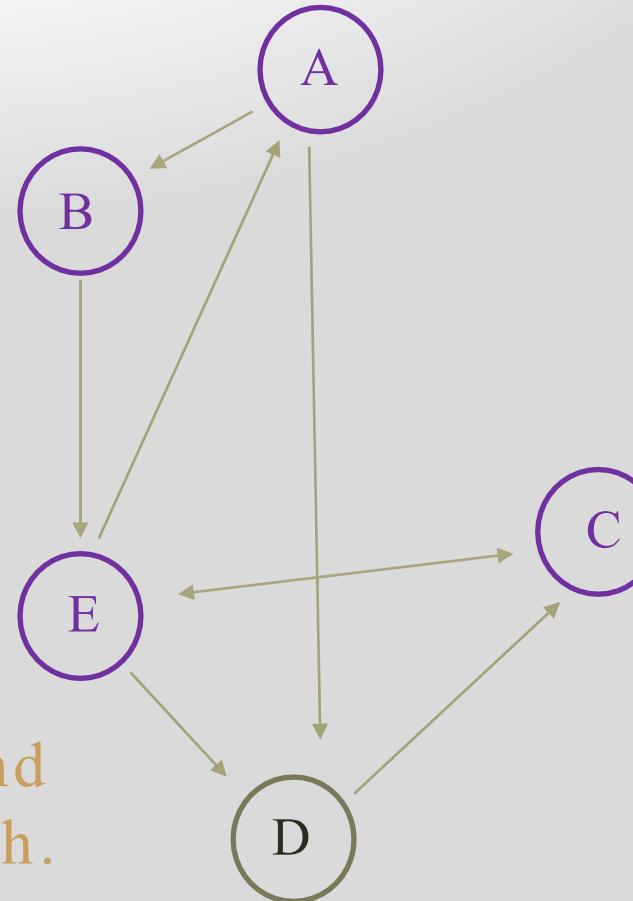
Path: A —B —E —C



Is there a better (shorter) path from A to C?



Is there a better (shorter) path from A to C?



DFS is not guaranteed to find the shortest path, just a path.

visited Not visited

```
DFS(Node curr, Node goal)
{
    curr.visited = true;
    if(curr.equals(goal))
        return;
    for(Node next : curr.neighbors)
        if(!next.visited)
        {
            next.cameFrom = curr;
            DFS(next, goal);
        }
}
// path is now saved in nodes' .cameFrom
```

# Summary

- Look at the first edge going out of the start node
- Recursively search from the new node
- Upon returning, take the next edge
- If no more edges, return

# Summary

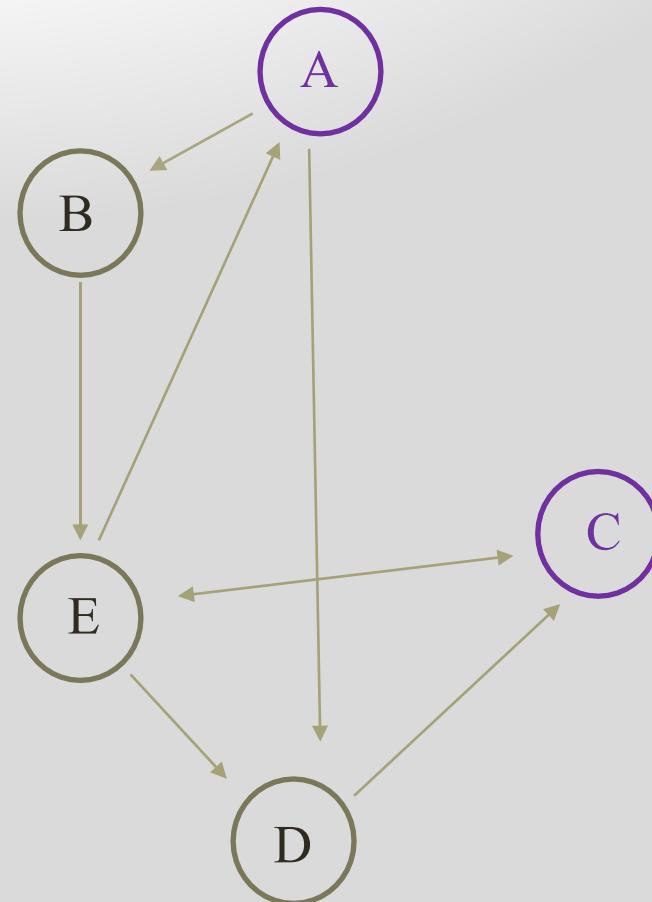
- Look at the first edge going out of the start node
  - Recursively search from the new node
  - Upon returning, take the next edge
  - If no more edges, return
- 
- When visiting a node, mark it as visited
    - So we don't get stuck in a cycle
    - Skip already visited nodes during traversal

# Summary

- Look at the first edge going out of the start node
- Recursively search from the new node
- Upon returning, take the next edge
- If no more edges, return
- When visiting a node, mark it as visited
  - So we don't get stuck in a cycle
  - Skip already visited nodes during traversal
- For each node visited, save a reference to the node where we came from to reconstruct the path

# Breadth-first search

We want to find a path from A to C

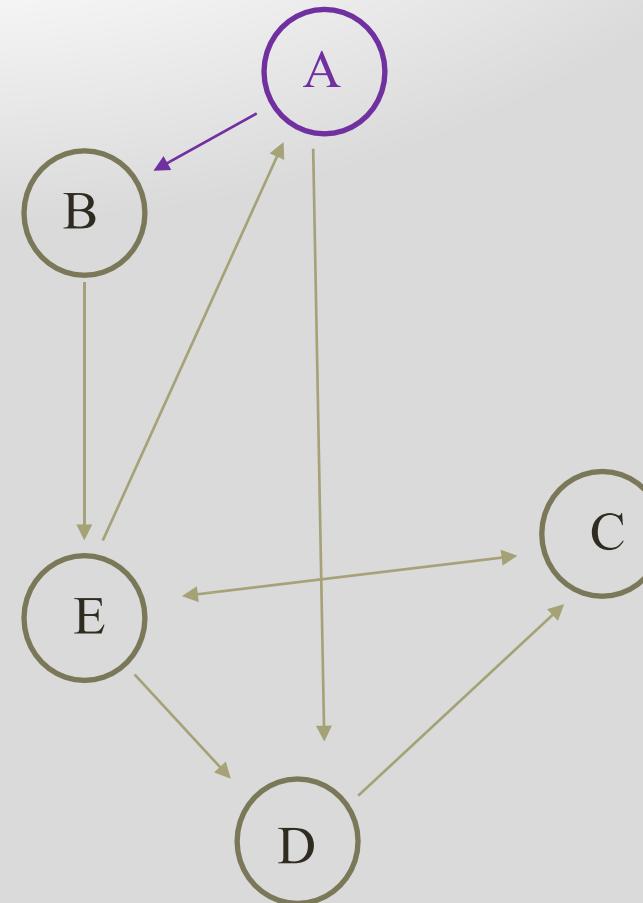


We want to find a path from A to C

Mark and enqueue the start node A

A.visited = true

Queue



visited



Not visited

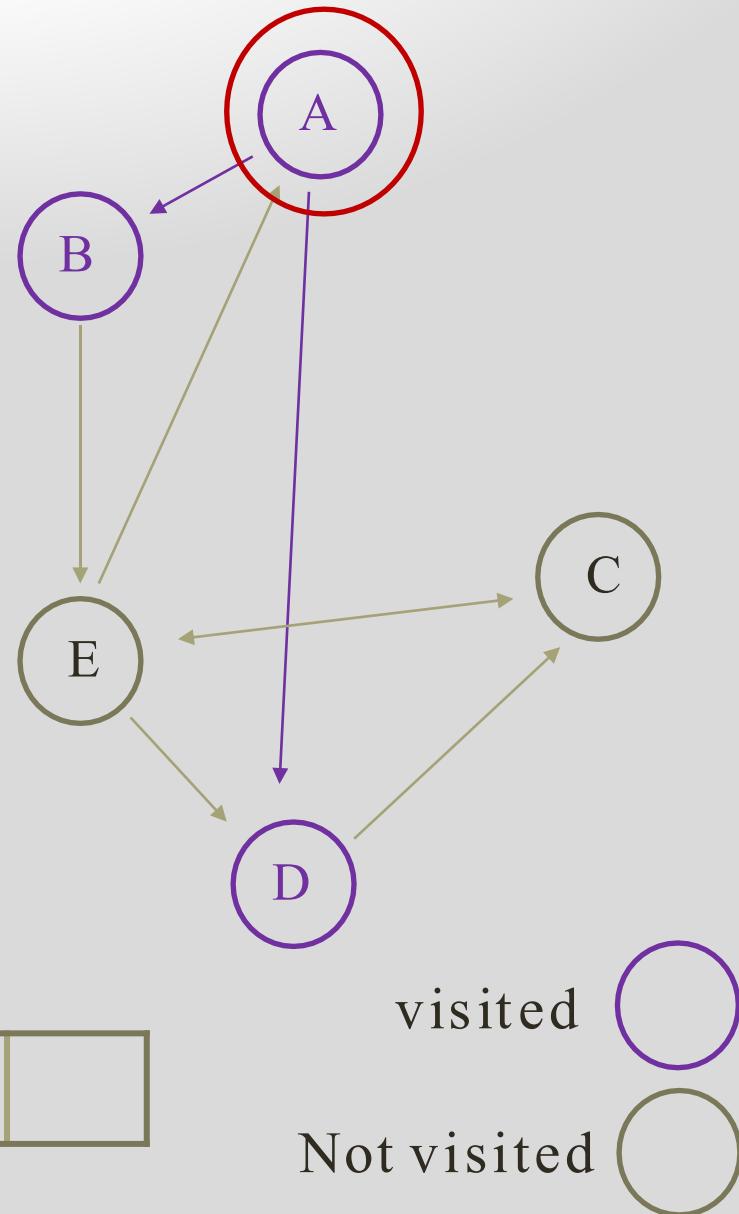


Dequeue the first node in the queue (A)

Mark and enqueue A's unvisited neighbors

```
B.cameFrom = A  
D.cameFrom = A  
B.visited = true  
D.visited = true
```

Queue [ B | D | ]

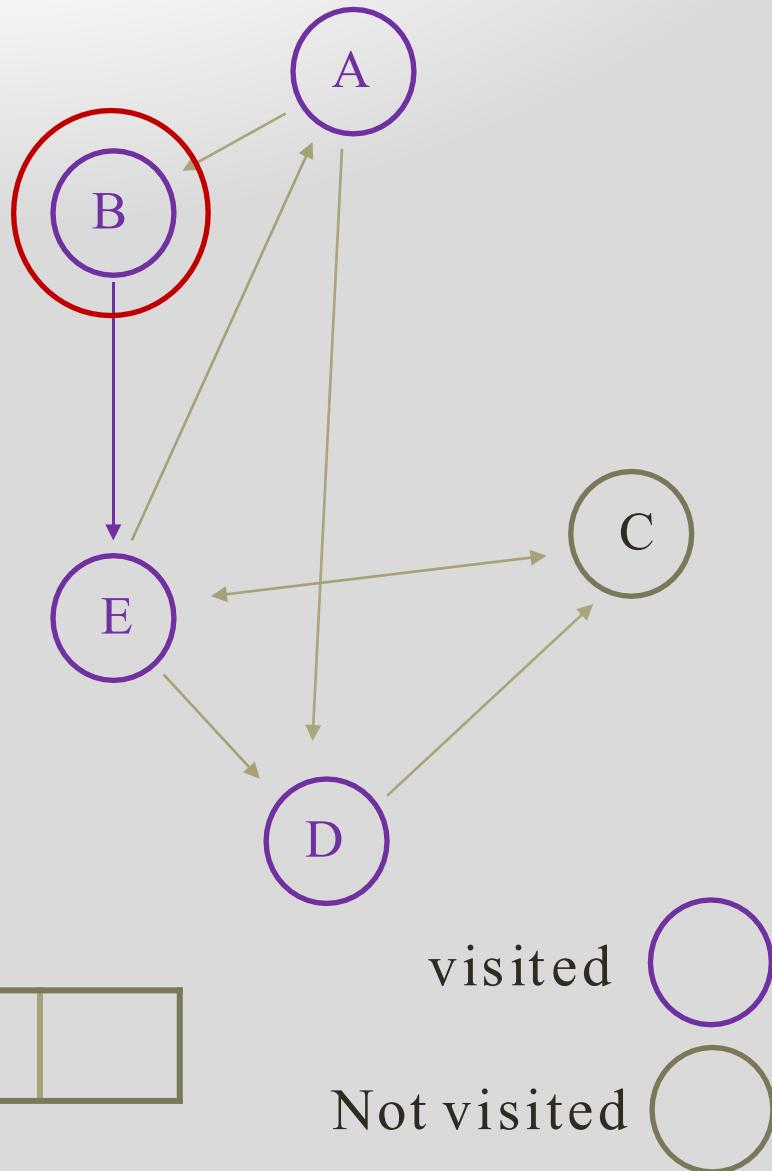


Dequeue the first node in the queue (B)

Mark and enqueue B's unvisited neighbors

```
E.cameFrom = B  
E.visited = true
```

Queue [ D | E | ]

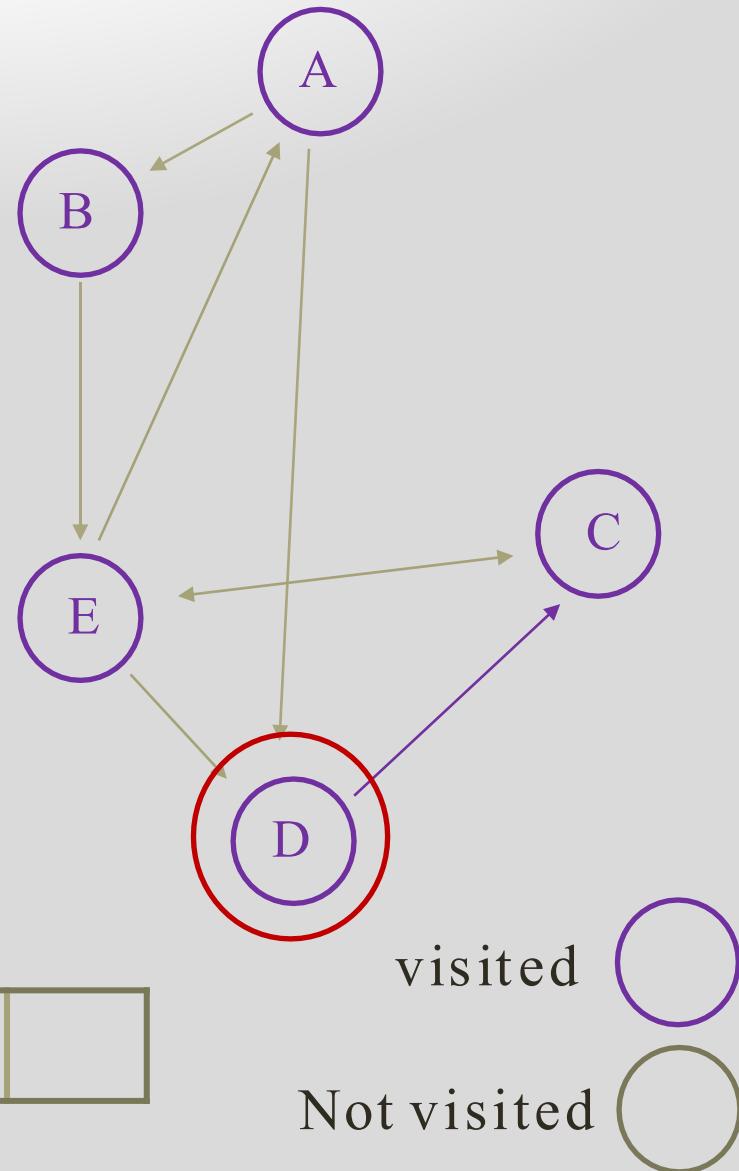


Dequeue the first node in the queue (D)

Mark and enqueue D's unvisited neighbors

C.cameFrom = D  
C.visited = true

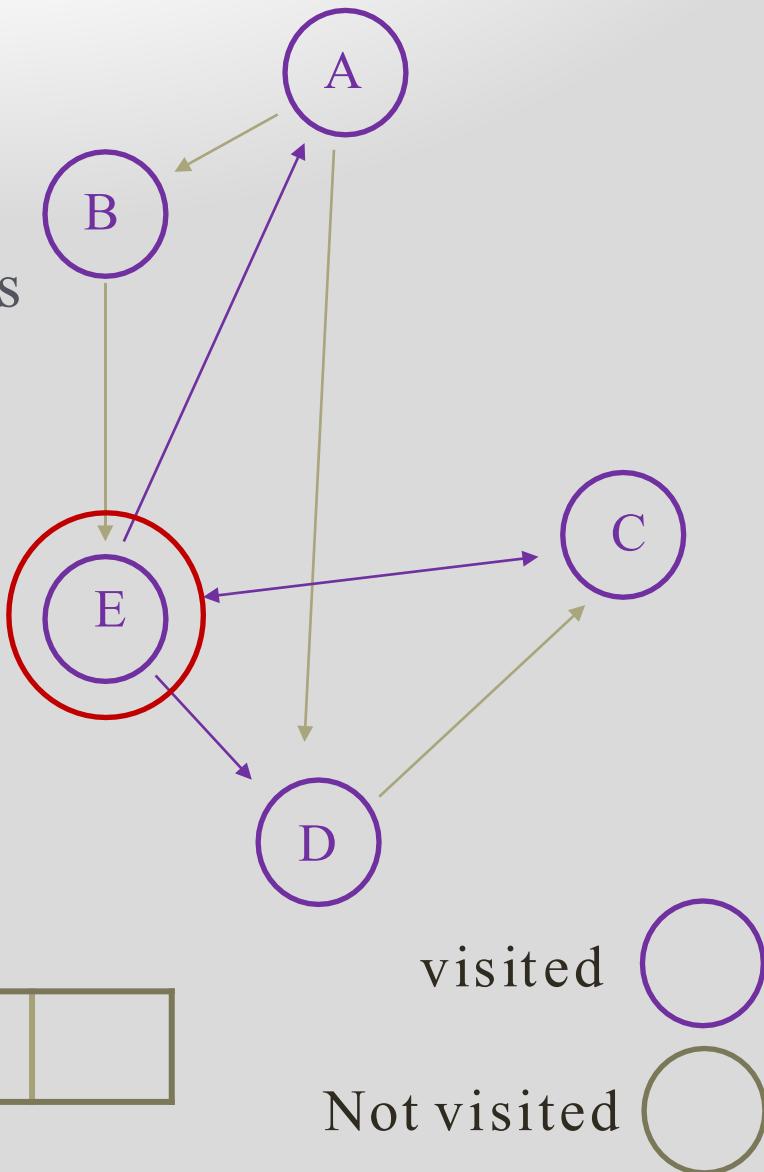
Queue [ E | C | ]



Dequeue the first node in the queue (E)

Mark and enqueue E's  
unvisited visited neighbors

(no unvisited neighbors!)



Queue [ C | ]

visited  
Not visited

[ 81 ]

Dequeue the first node in the queue (C)

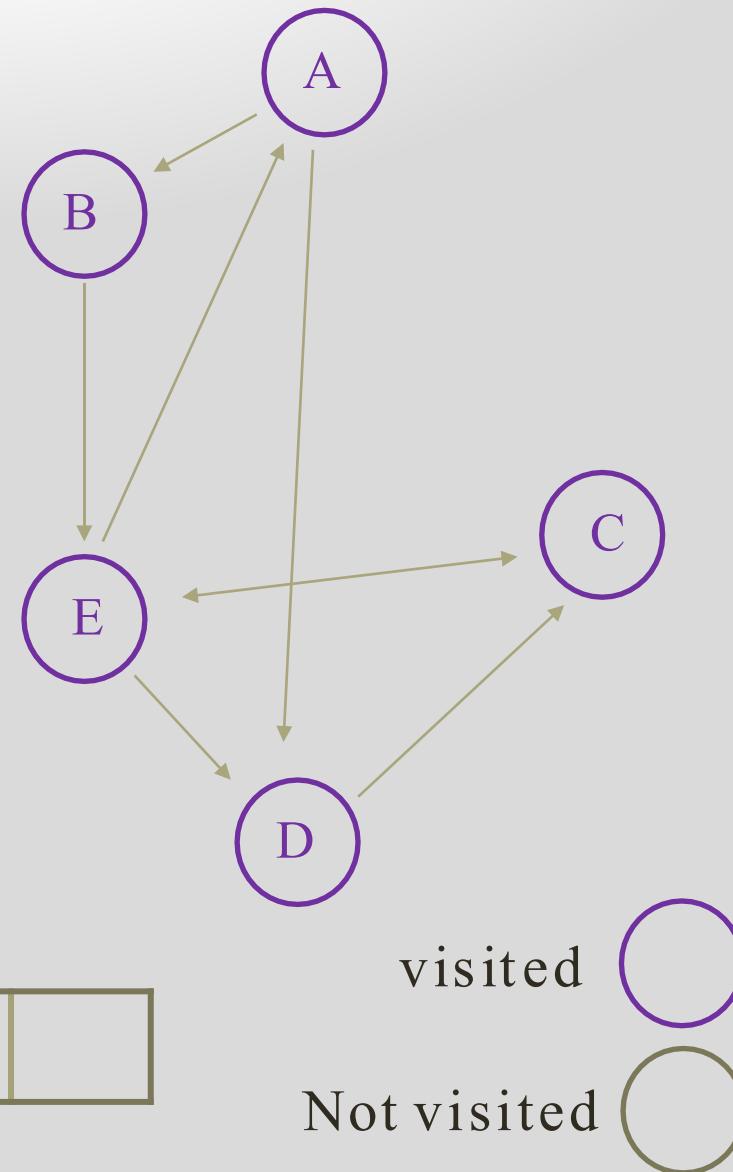
C is the goal! Reconstruct  
the path visited  
with **cameFrom** references

C.cameFrom = D

D.cameFrom = A

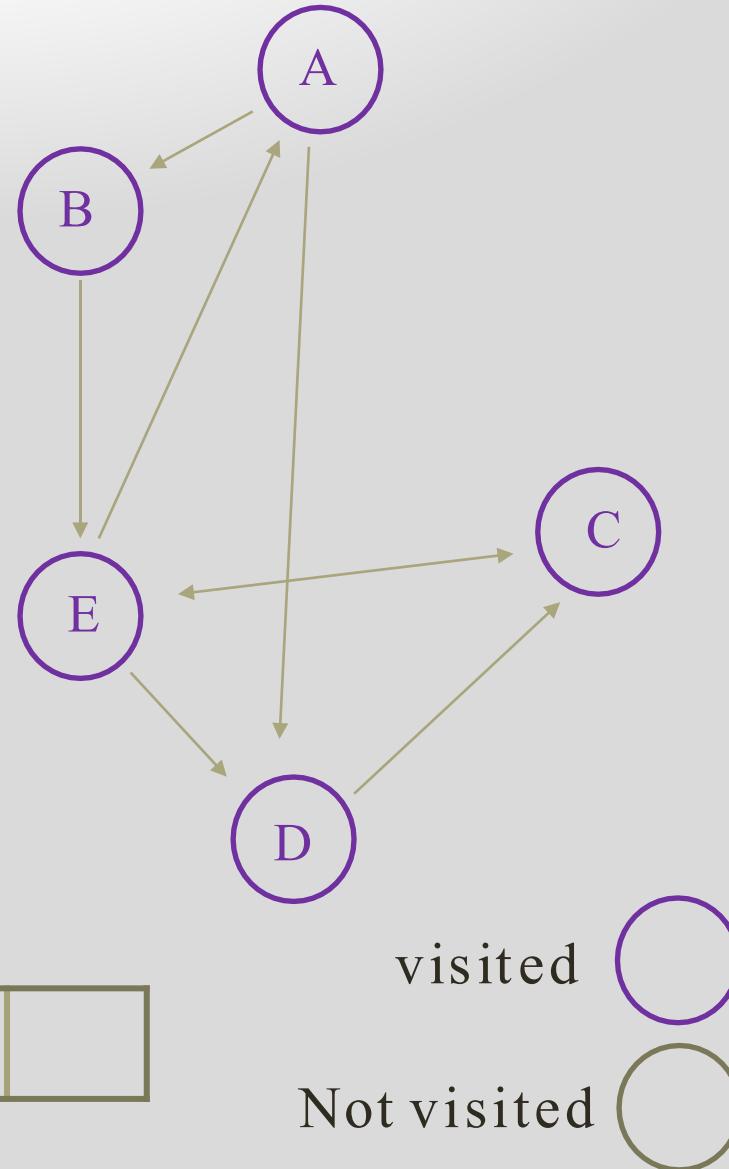
Path: A —D —C

Queue



Is this the shortest path?

Path: A — D — C



Queue



visited



Not visited



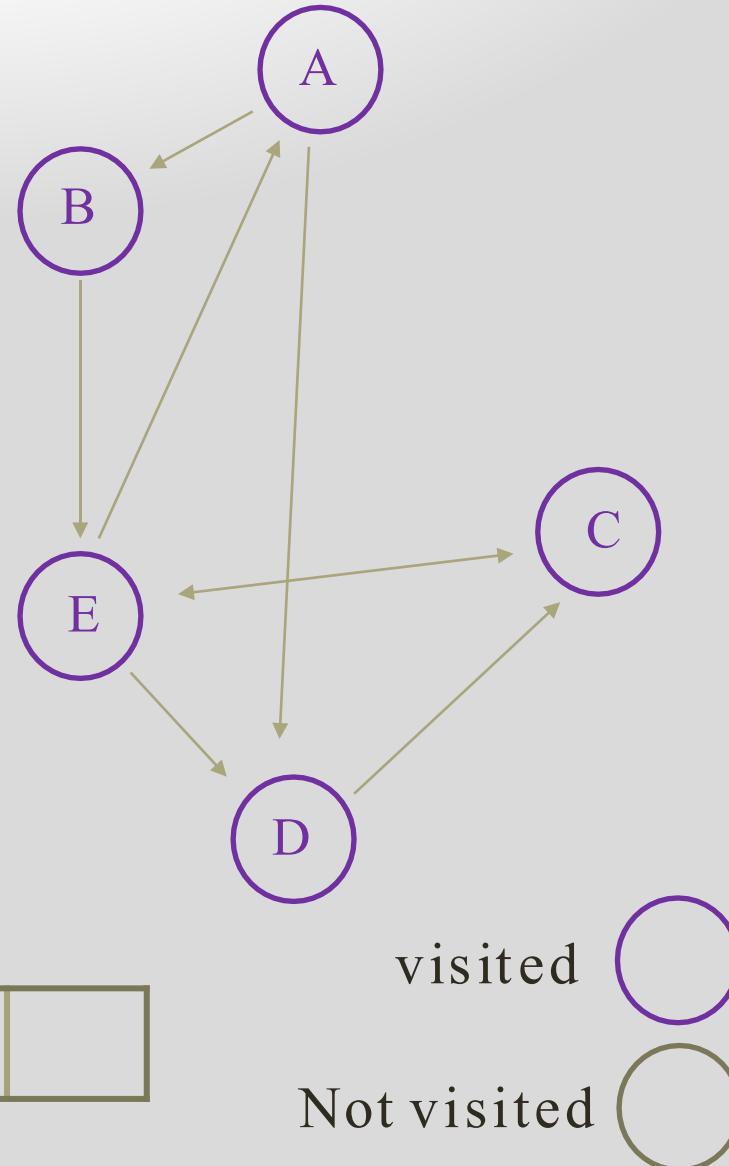
Is this the shortest path?

Path: A — D — C

BFS visits nodes closest to the start-point first

Therefore, the first path found is the shortest path (closest to the start node)

Queue



visited



Not visited

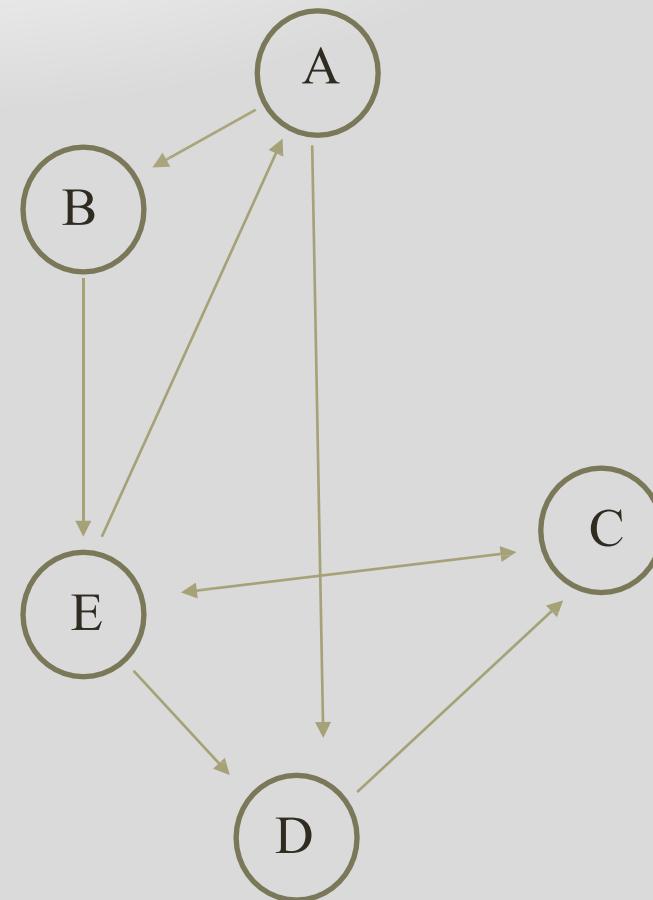


```
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)
    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return
        for(Node next : curr.neighbors)
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
```

- Instead of visiting deeper nodes first, visit shallower nodes first
  - Visit nodes closest to the start point first, gradually get further away
- Create an empty queue
- Put the starting node in the queue
- While the queue is not empty
  - Dequeue the current node
  - For each unvisited neighbor of the current node
    - *Mark the neighbor as visited*
    - *Put the neighbor into the queue*
- Notice it is not recursive... it just runs until the queue is empty!

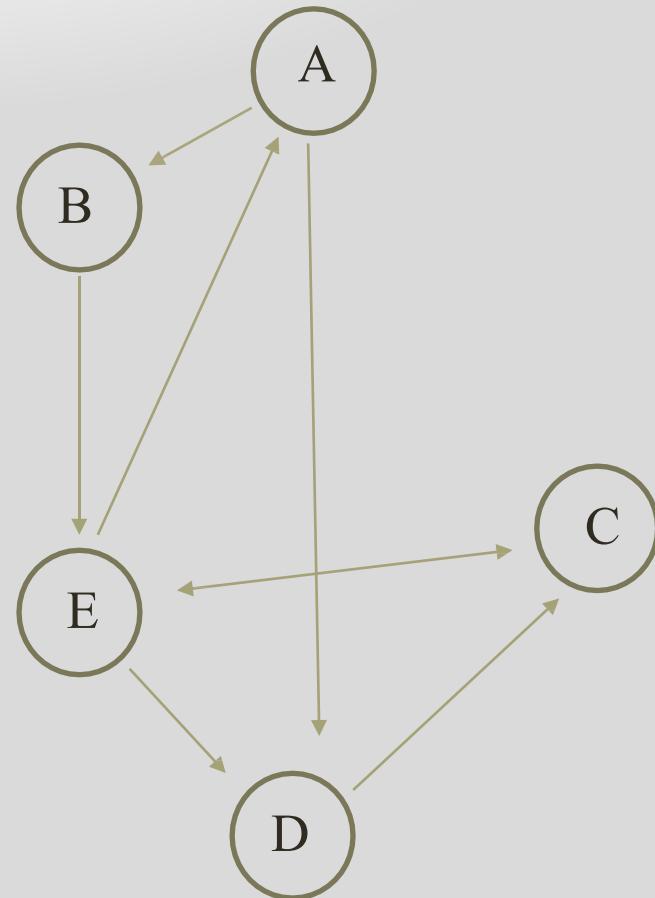
# What path will BFS find from B to C?

- A) B E C
- B) B E A D C
- C) B E D C
- D) none



# What path will DFS find from A to D?

- A) A B E D
- B) AD
- C) none
- D) this is a trick question



# **What is true of DFS, searching from a start node to a goal node?**

- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a and b
- E) a and c
- F) a, b, and c

# **What is true of BFS, searching from a start node to a goal node?**

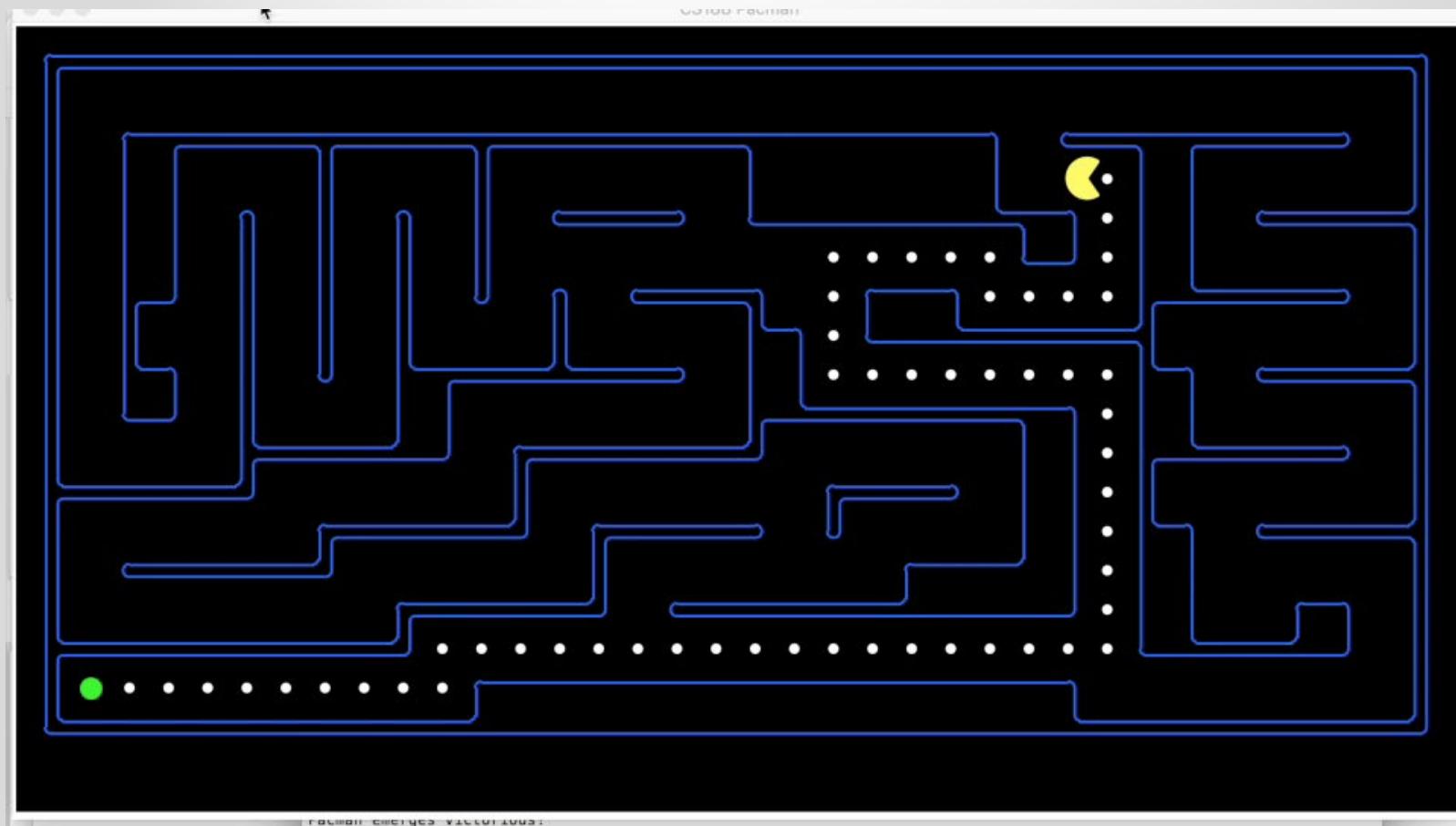
- A) If a path exists, it will find it
- B) It is guaranteed to find the shortest path
- C) It is guaranteed to not find the shortest path
- D) a, and b
- F) a, and c

# Summary

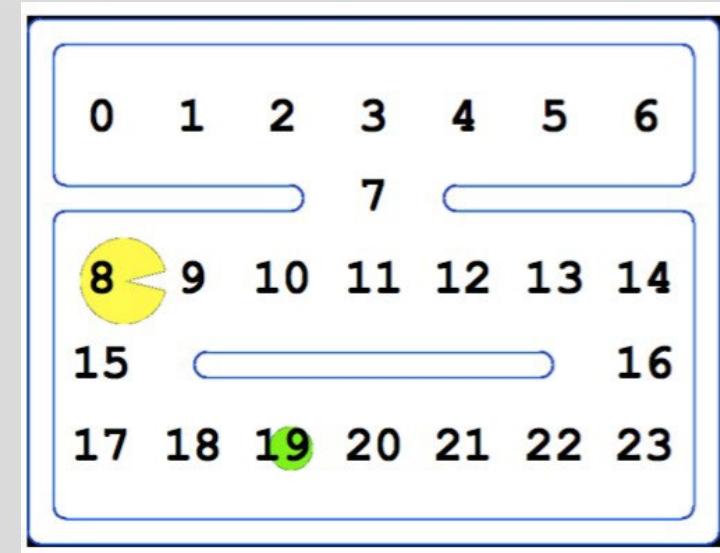
- Depth-first search — DFS
- Breadth-first search — BFS
- If there exists a path from one node to another these algorithms will find it
  - The nodes on this path are the steps to take to get from point A to point B
- If multiple such paths exist, the algorithms may find different ones

Today...

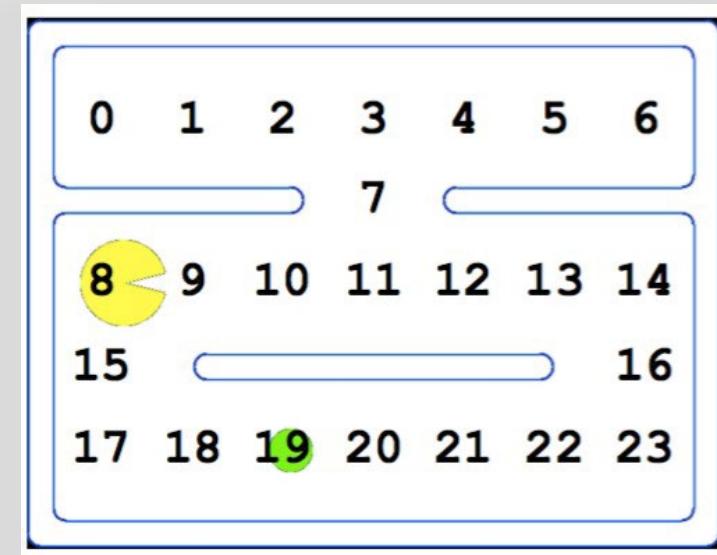
- BFS and the homework
- Weighted graphs
- Dijkstra's algorithm



- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue

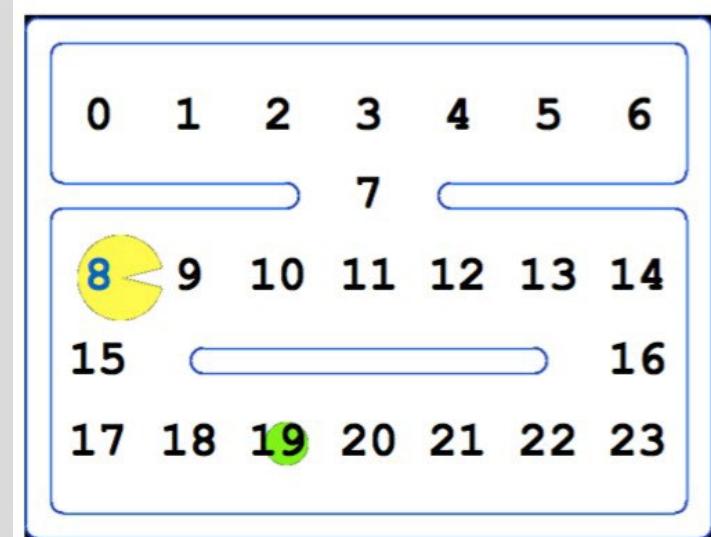


Current:

Queue:

visited      ●  
Came from ←

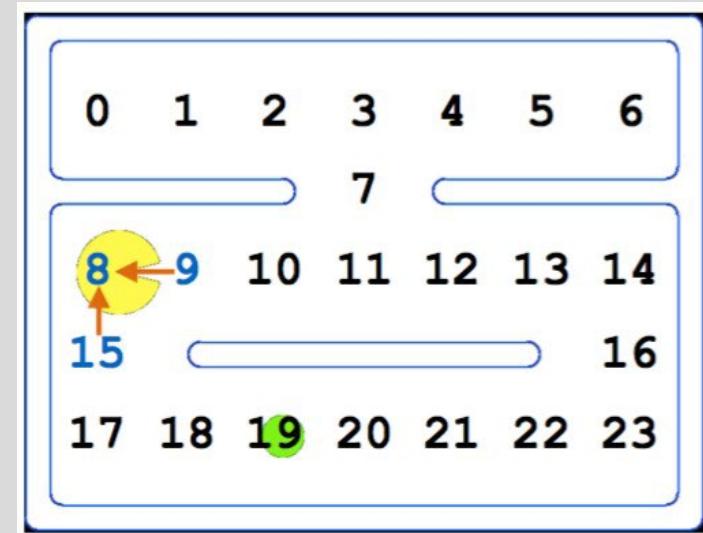
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current:  
Queue: 8

visited  
Came from ←

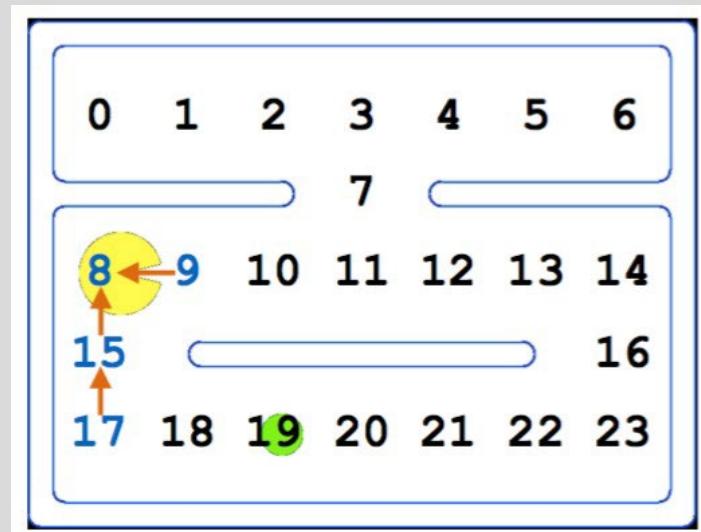
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 8  
Queue: 15 9

visited      ●  
Came from ←

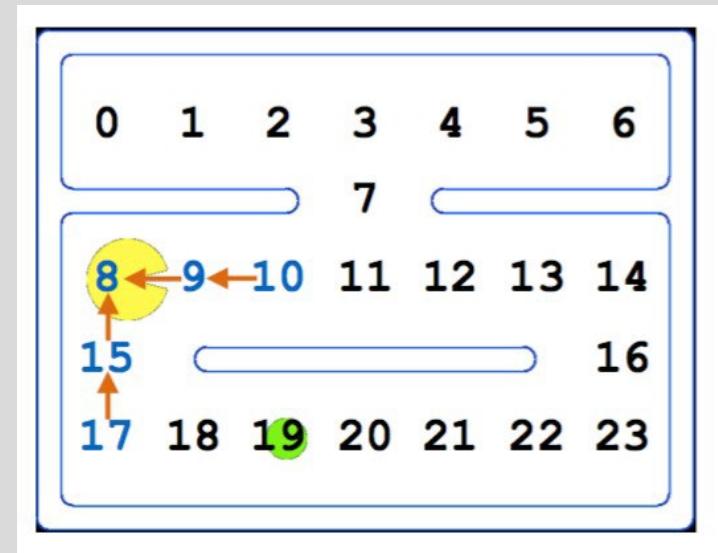
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 15  
Queue: 9 17

visited      ●  
Came from ←

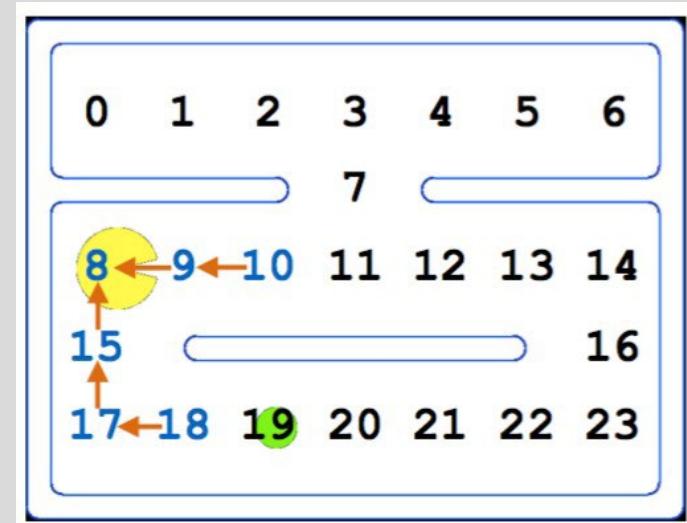
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 9  
Queue: 17 10

visited      ●  
Came from ←

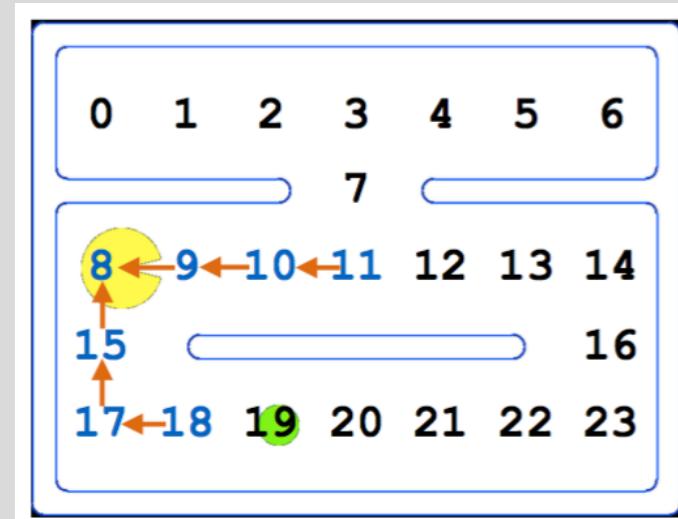
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 17  
Queue: 10 18

visited      ●  
Came from ←

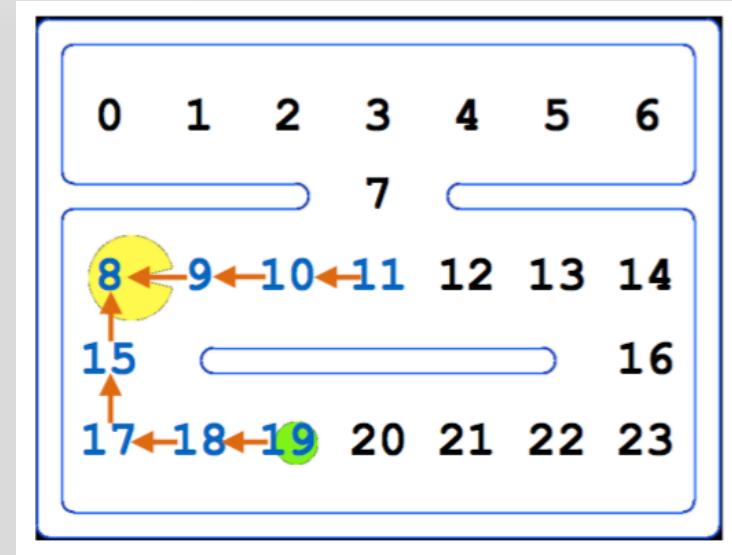
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 10  
Queue: 18 11

visited      ●  
Came from ←

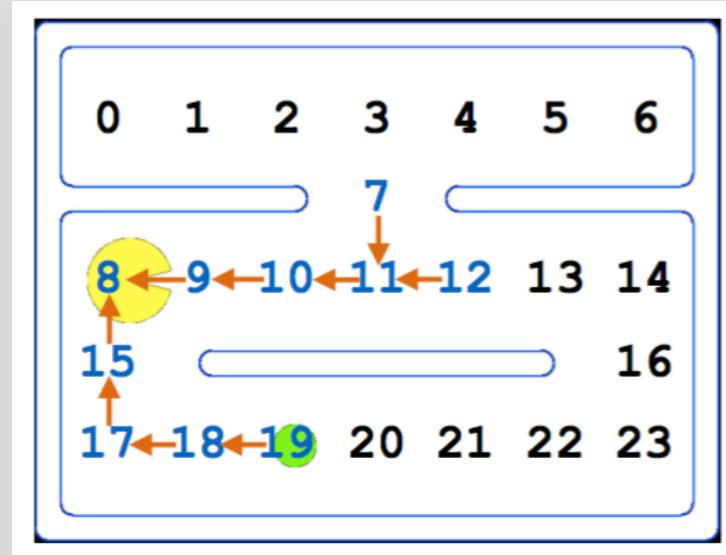
- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



Current: 18  
Queue: 11 19

visited      ●  
Came from ←

- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



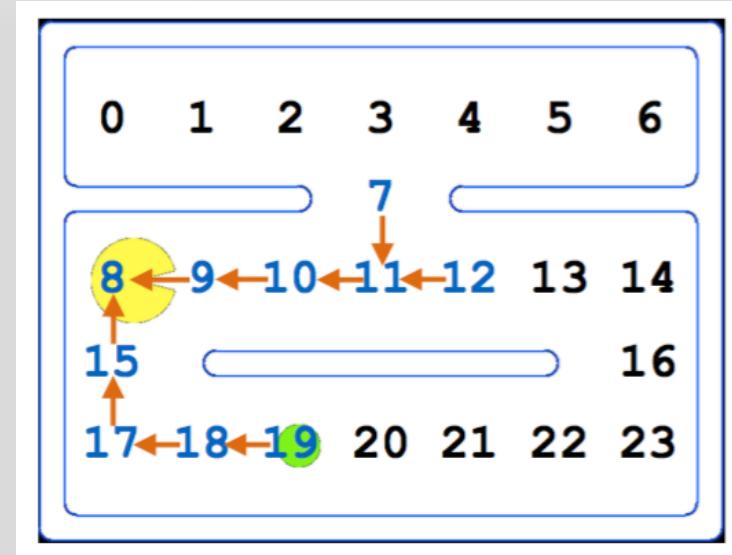
Current: 11

Queue: 19 7 12

visited

Came from ←

- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



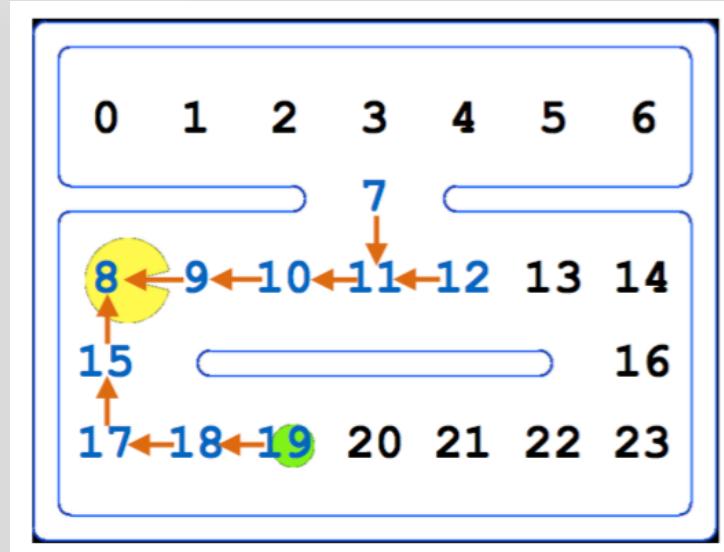
Current: 19 **GOAL!**

Queue: 7 12

visited

Came from ←

- Put the starting node in the queue and mark it as visited
- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue



reconstruct the path...

Current: 19 **GOAL!**

Queue: 7 12

visited

Came from ←

- The problem:
  - Input: a simple text file describing the maze
    - *Includes wall locations, start point, and end point*
  - Output: a similar text file with the shortest path from start to end indicated in the maze
- Represent all possible moves with a graph, then do a breadth first search

input

```
5 10
XXXXXXXXXX
X S       X
X           X
X       G   X
XXXXXXXXXX
```

output

```
5 10
XXXXXXXXXX
X S..... X
X       .   X
X       G   X
XXXXXXXXXX
```

**X** wall segment

**S** starting point

**G** goal

an open space

• solution path indicator

input

```
5 10
XXXXXXXXXX
X S       X
XXXXXXX X
X G       X
XXXXXXXXXX
```

output

```
5 10
XXXXXXXXXX
X S.....X
XXXXXXX.X
X G.....X
XXXXXXXXXX
```

**X** wall segment

**S** starting point

**G** goal

an open space

• solution path indicator

input

```
10 19
XXXXXXXXXXXXXXXXXXXXXX
X      S      X
X          X
X          X
XX X      X      X
X X XXX      X      X
X G      XX      X
X          X
X          X
XXXXXXXXXXXXXXXXXXXXXX
```

output

```
10 19
XXXXXXXXXXXXXXXXXXXXXX
X      S.      X
X          .
X          .
XX X      . X      X
X X XXX      . X      X
X G....XX.
X      .....
X
XXXXXXXXXXXXXXXXXXXXXX
```

**X** wall segment

**S** starting point

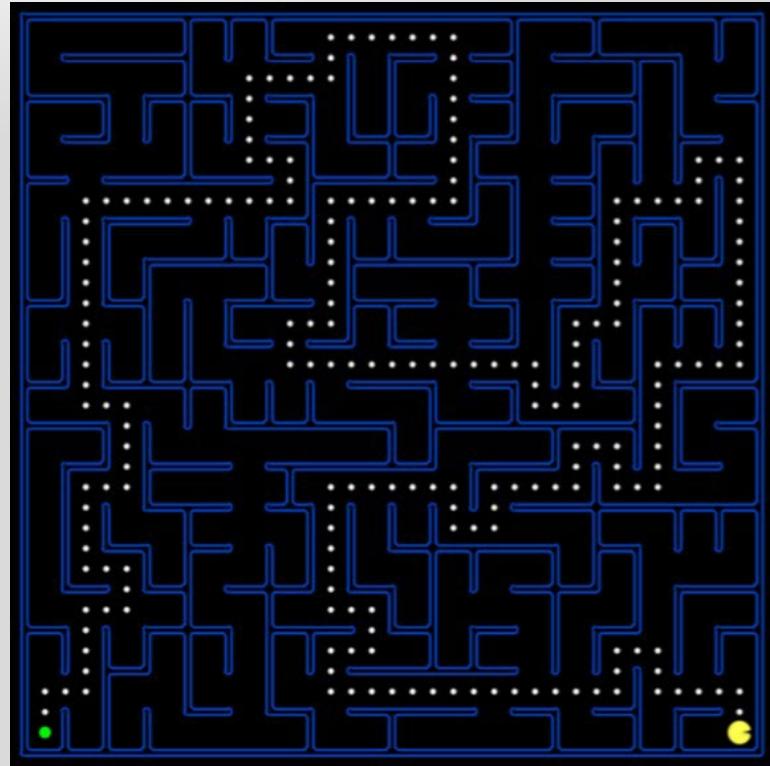
**G** goal

an open space

.

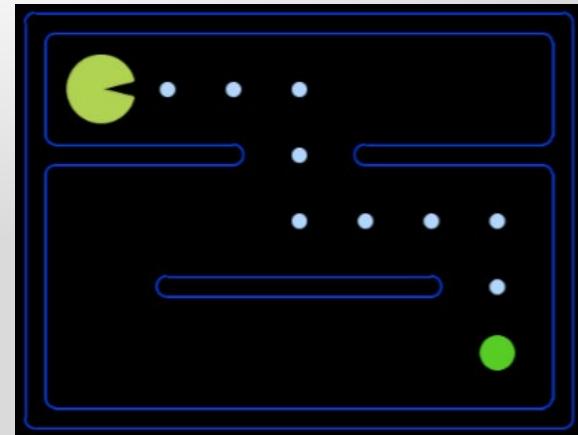
solution path indicator

- Can any node have an edge to any other node?
- How do we represent walls?



- For this specific problem, we can store the graph as a 2D array

```
Node nodes [ ] [ ] ;  
nodes = new Node [ 5 ] [ 10 ] ;
```



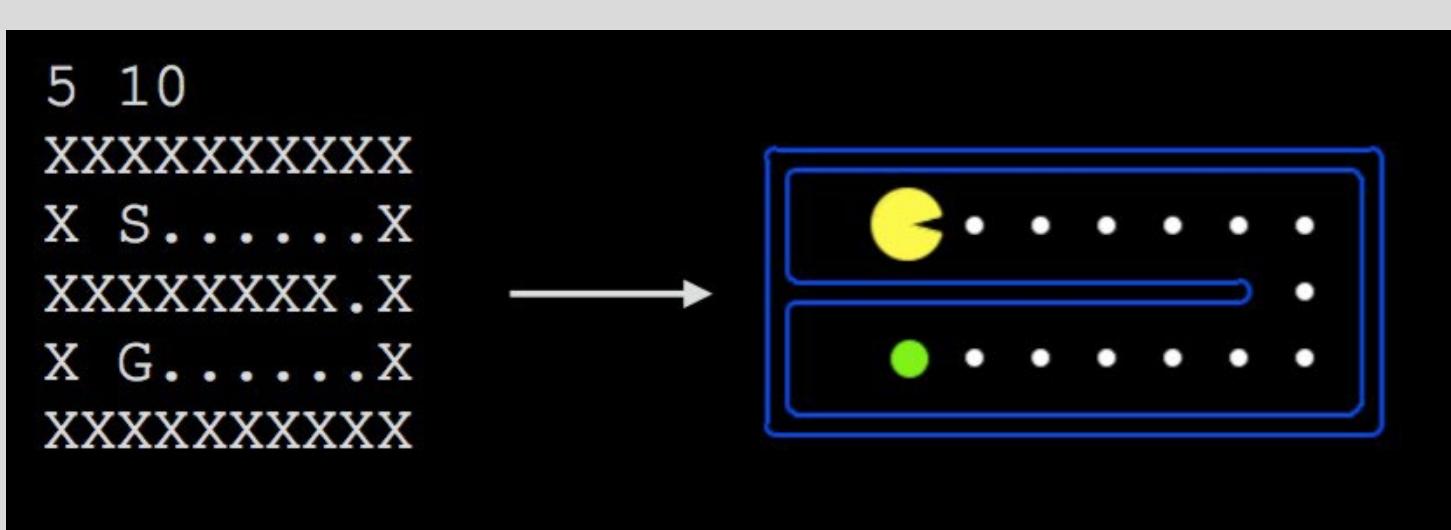
- Node class doesn't need a list of edges
  - Neighbors are implied (up, down, left right)
  - i.e., for node N, the up neighbor would be: `nodes[N.row-1][N.col]`
- Walls are null
  - i.e., no neighbor if null

- While reading the input:
  - For every character that makes up the maze [i][j]  
*if it is a wall*  
    nodes[i] [j] = null  
*else*  
    nodes[i] [j] = new Node(...)
- Make sure to handle the start and goal nodes

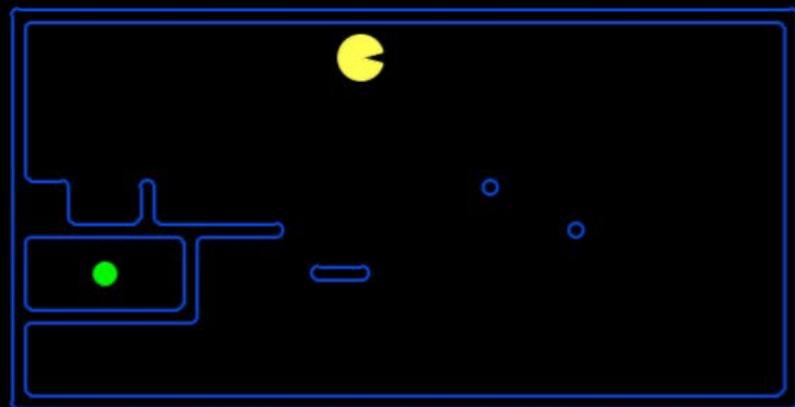
# Rules

- The path cannot go through or on top of walls
- The path must be connected (no skips or jumps)
- Diagonally-adjacent spaces are not connected
  - Only up, down, left, right
- If no path exists, the output file will have no dots
- If multiple shortest paths exist, any of them are valid
- **Must produce output in exact format specified**

- All you have to do is read in a file and produce a new file
- BUT, we are providing a program to read in your solution and display it as a pacman game board



```
XXXXXXXXXXXXXXXXXXXX  
X      S      X  
X          X  
X          X  
XX X      X      X  
XXXXXXX      X      X  
X G X  XX      X  
XXXXX      X  
X          X  
XXXXXXXXXXXXXXXXXXXX
```



# File output

```
try
{
    PrintWriter output = new PrintWriter(new
        FileWriter("example.txt"));

    output.print("G");
    output.print("X");
    output.println();
    ouput.close();

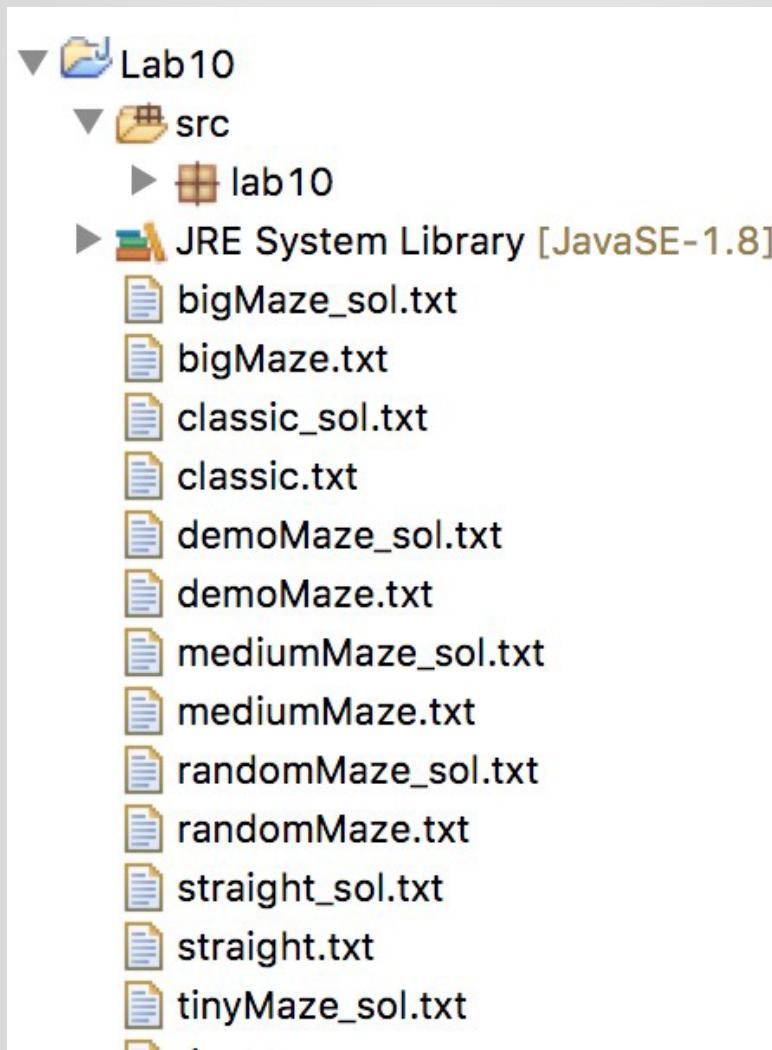
}
```

file will contain “GX” and a newline

# Reading numbers

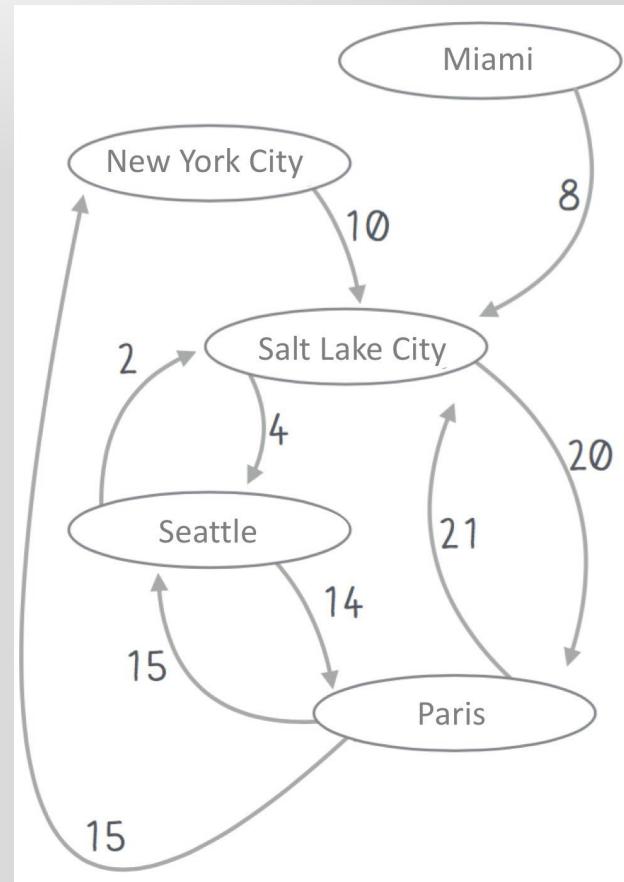
```
int height, width;  
  
String[] dimensions =  
    input.readLine().split(" ");  
  
try  
{  
    height = Integer.parseInt(dimensions[0]);  
    width = Integer.parseInt(dimensions[1]);  
}
```

# File handling

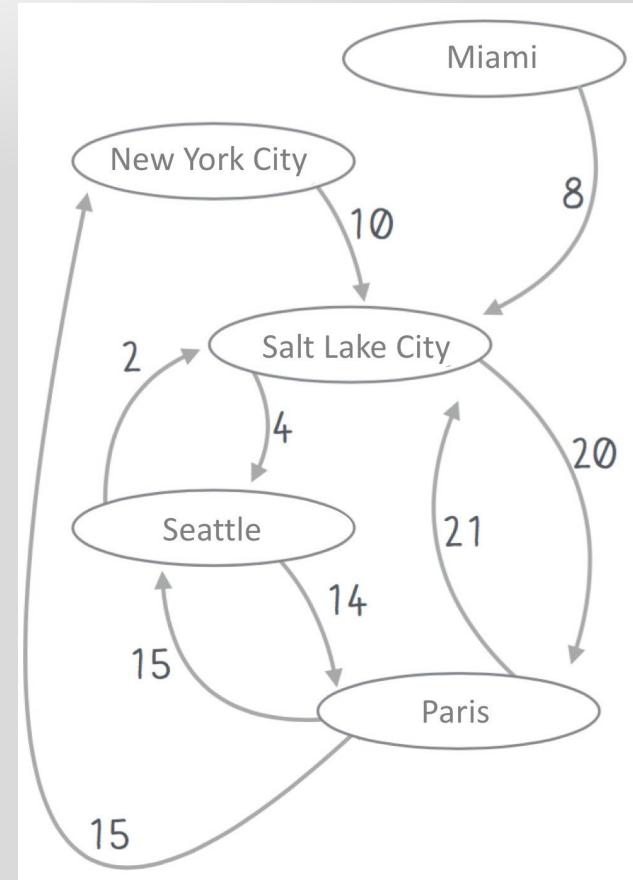


# Weighted graphs

- Sometimes it makes sense to associate a cost with traversing an edge
- We can add a **weight** to each edge
  - This is just a number!
- A higher weight indicates a more costly step
- **Weighted path length** is the sum of all edge weights on a path
  - this is ***NOT*** the same as path length!



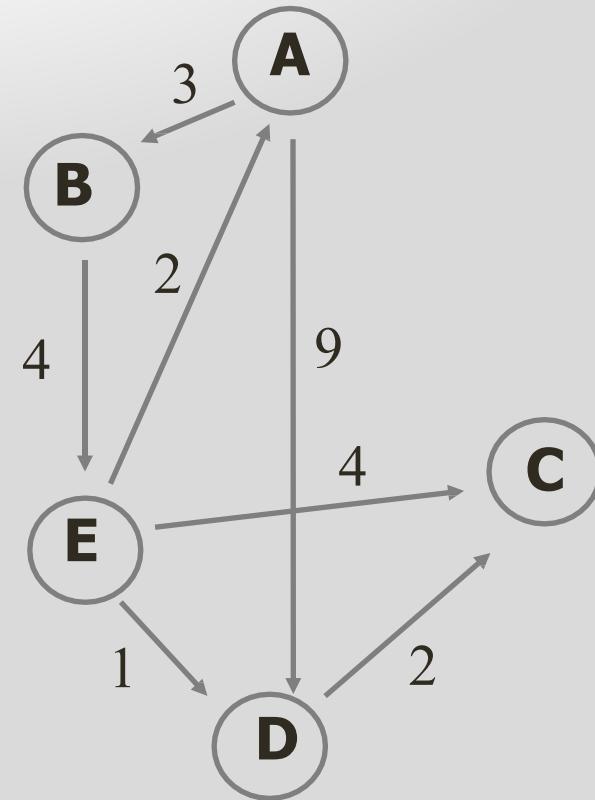
- What is the *shortest* path from MIA to Paris?
- What is the *cheapest* path from MIA to Paris?
- Cheapest is not always the shortest!
- Will regular BFS find the cheapest path?



# Dijkstra's algorithm

- Dijkstra's algorithm finds the *cheapest* path
- Keep track of the total path cost from start node to the current node
- Cost of path to next node is total cost so far plus weight of edge to next node
- Instead of traversing nodes in the order they were encountered, traverse in order of cheapest total cost first

- We want to find a path from A to C



Priority queue



visited



( 76 )

unvisited

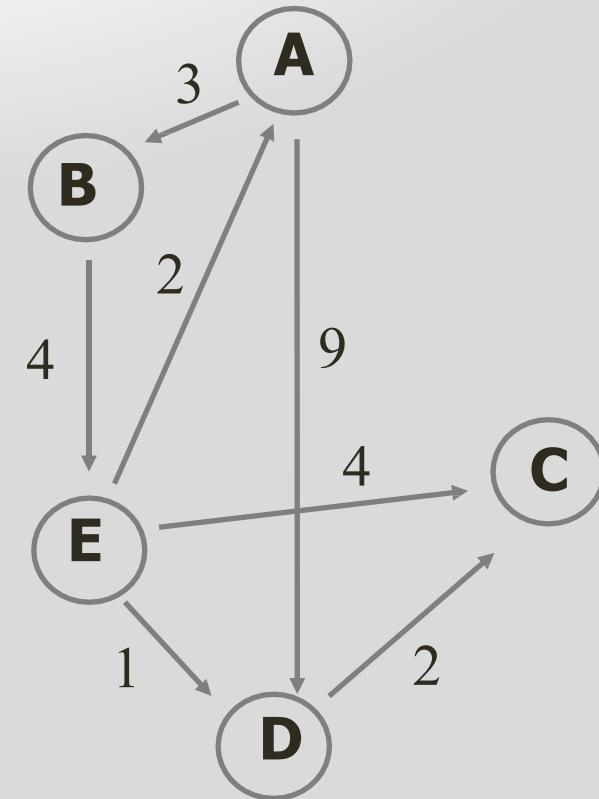


- We want to find a path from A to C

- This time we use a priority queue.

Mark nodes **after** removal from the queue.

Priority queue



visited



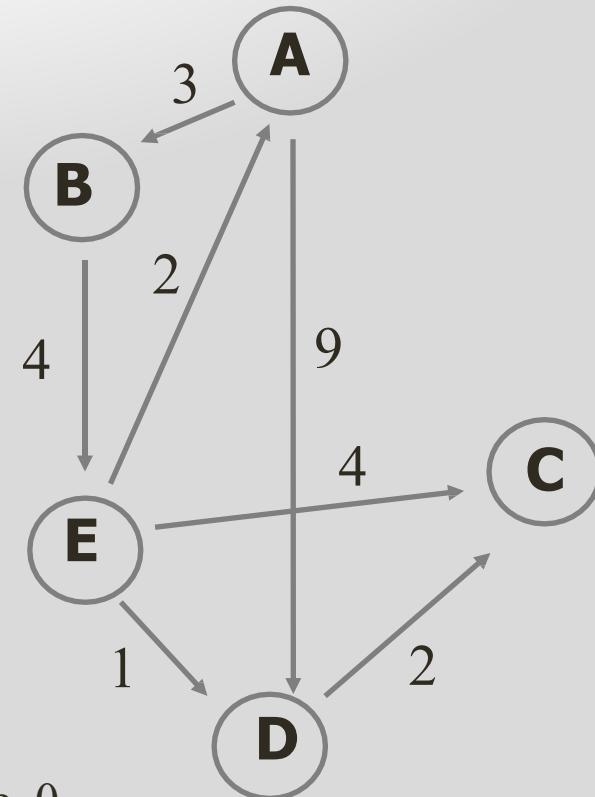
( 77 )

unvisited



- We want to find a path from A to C

A.costSoFar = 0



0 is smaller than the largest integer, so use 0

Priority queue

|      |  |  |  |
|------|--|--|--|
| A(0) |  |  |  |
|------|--|--|--|

visited



( 78 )

#(\*): # indicates a node, \* indicate the smallest cost to node #. Initialize all # to largest integer.

unvisited



- we want to find a path from A to C

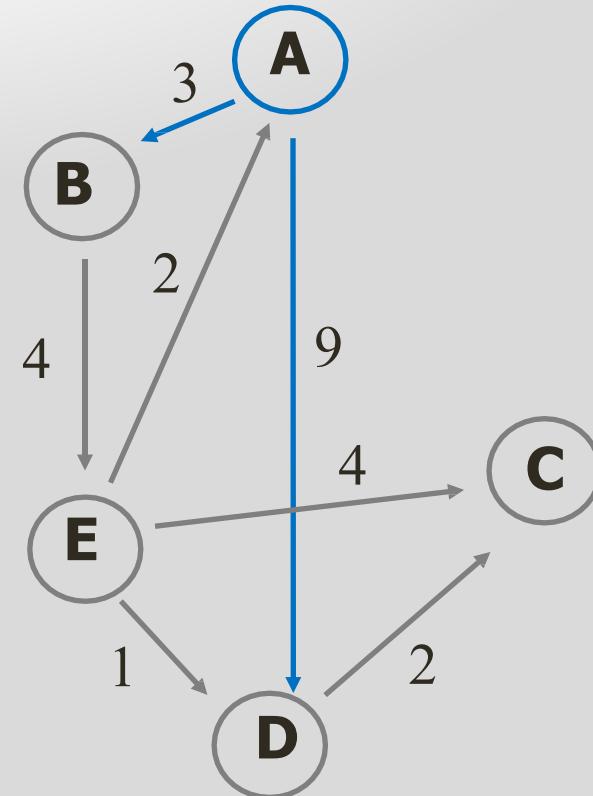
Dequeue A(0), and enqueue A's neighbors with A's cost-so-far plus the edge weight

B.costSoFar = A.costSoFar + 3

D.costSoFar = A.costSoFar + 9

B.cameFrom = A

D.cameFrom = A



Priority queue

|      |      |  |  |
|------|------|--|--|
| B(3) | D(9) |  |  |
|------|------|--|--|

visited



( 79 )

3 and 9 both are smaller than the largest integer, unvisited so use 3 and 9 as the distances to nodes B and D.

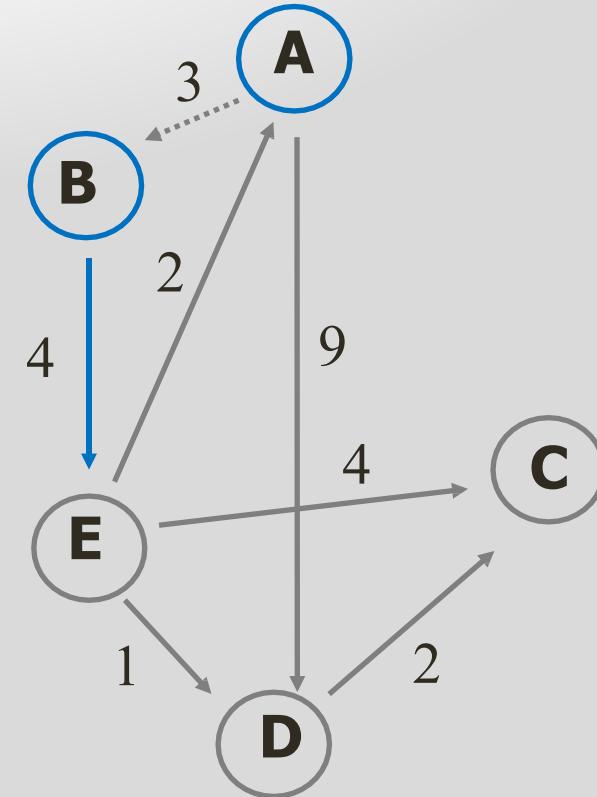


- we want to find a path from A to C

Dequeue B(3), and enqueue B's neighbors with B's cost-so-far plus the edge weight

`E.costSoFar = B.costSoFar + 4`

`E.cameFrom = B`



Priority queue

|      |      |  |  |
|------|------|--|--|
| E(7) | D(9) |  |  |
|------|------|--|--|

visited



( 80 )

E's current (initialized) distance is the largest integer, and 7 is smaller than the largest integer, so use 7



- we want to find a path from A to C

Dequeue E(7), and enqueue E's neighbors with E's cost-so-far plus the edge weight

```
// A visited, so skip
// cheaper path to D found!
C.costSoFar = E.costSoFar + 4
D.costSoFar = E.costSoFar + 1
D.cameFrom = E
```

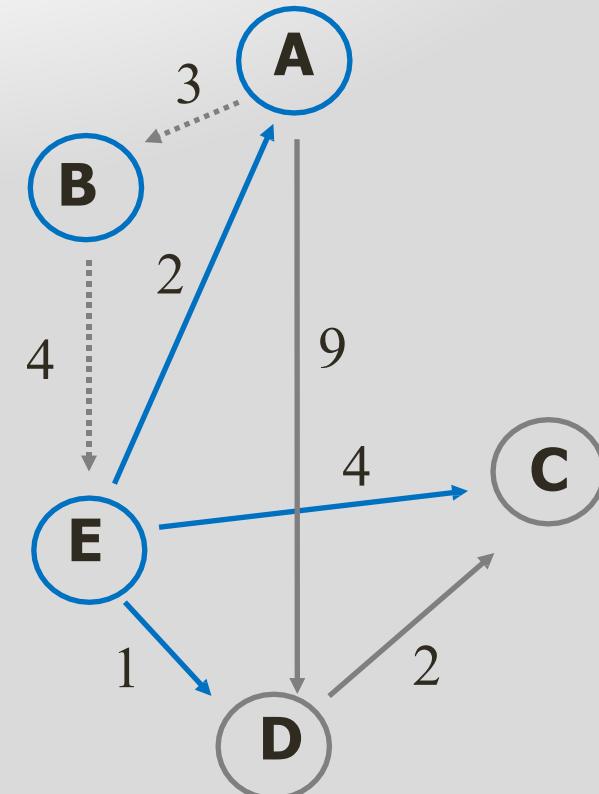
Priority queue

|      |       |  |  |
|------|-------|--|--|
| D(8) | C(11) |  |  |
|------|-------|--|--|

visited



( 81 )



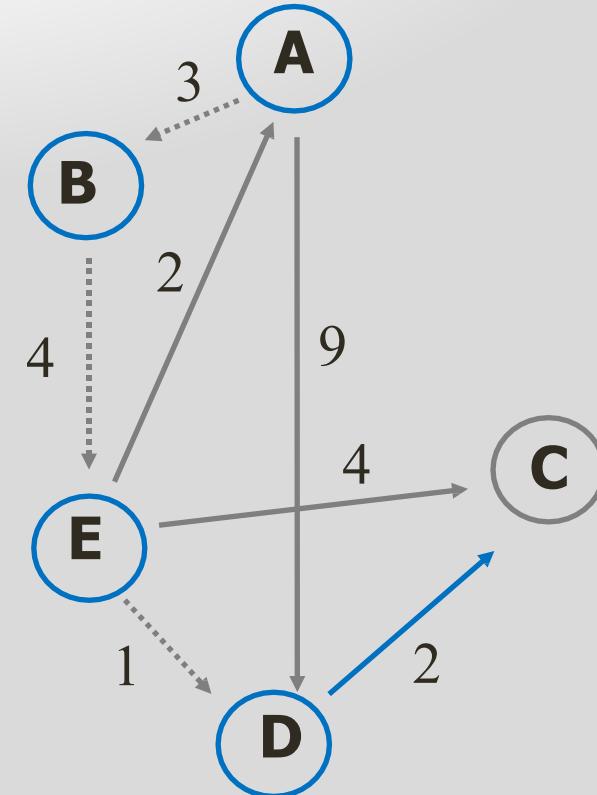
D's previous distance is 9, now it is 8, so use 8, unvisited and also change its cameFrom attribute.



- we want to find a path from A to C

Dequeue D(8), and enqueue D's neighbors with D's cost-so-far plus the edge weight

```
// cheaper path to C found!
C.costSoFar = D.costSoFar + 2
C.cameFrom = D
```



Priority queue

|       |  |  |  |
|-------|--|--|--|
| C(10) |  |  |  |
|-------|--|--|--|

visited



unvisited

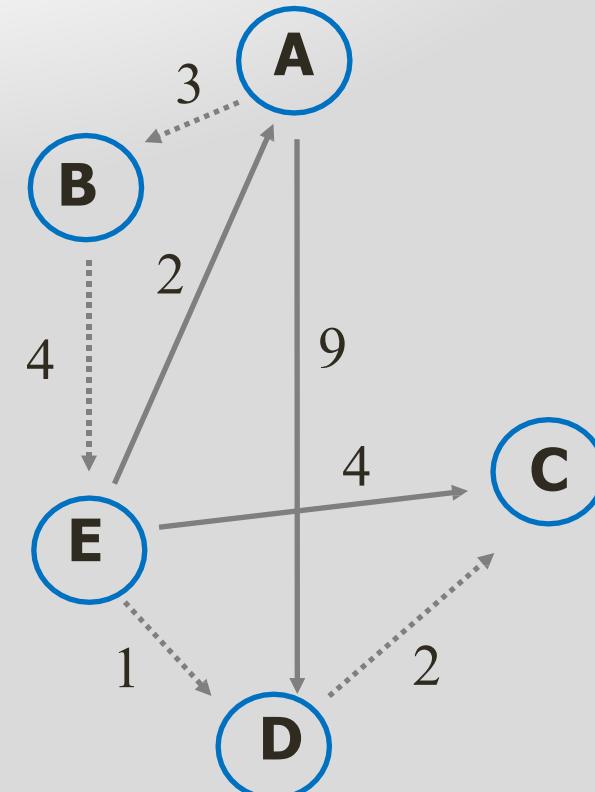


- we want to find a path from A to C

Dequeue C(10). We found our goal! Final cost is 10.

Reconstruct path.

A—B—E—D—C



Priority queue



visited



( 83 )

unvisited



```

1 // Java implementation of Dijkstra's Algorithm
2 // using Priority Queue
3 import java.util.*;
4 public class DPQ {
5     private int dist[];
6     private Set<Integer> settled;
7     private PriorityQueue<Node> pq;
8     private int V; // Number of vertices
9     List<List<Node>> adj;
10
11    public DPQ(int V)
12    {
13        this.V = V;
14        dist = new int[V];
15        settled = new HashSet<Integer>();
16        pq = new PriorityQueue<Node>(V, new Node());
17    }
18
19    // Function for Dijkstra's Algorithm
20    public void dijkstra(List<List<Node>> adj, int src)
21    {
22        this.adj = adj;
23
24        for (int i = 0; i < V; i++)
25            dist[i] = Integer.MAX_VALUE;
26
27        // Add source node to the priority queue
28        pq.add(new Node(src, 0));
29
30        // Distance to the source is 0
31        dist[src] = 0;
32        while (settled.size() != V) {
33
34            // remove the minimum distance node
35            // from the priority queue
36            int u = pq.remove().node;
37
38            // adding the node whose distance is
39            // finalized
40            settled.add(u);
41
42            e_Neighbours(u);
43        }
44    }

```

- ArrayList maintains the order of the object in which they are inserted while HashSet is an unordered collection and doesn't maintain any order.
- ArrayList allows duplicate values while HashSet doesn't allow duplicates values.
- ArrayList is index based we can retrieve object by calling get(index) method or remove objects by calling remove(index) method while HashSet is completely object based. HashSet also does not provide get() method.

`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`

Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

```
45
46 // Function to process all the neighbours
47 // of the passed node
48 private void e_Neighbours(int u)
49 {
50     int edgeDistance = -1;
51     int newDistance = -1;
52
53     // All the neighbors of v
54     for (int i = 0; i < adj.get(u).size(); i++) {
55         Node v = adj.get(u).get(i);
56
57         // If current node hasn't already been processed
58         if (!settled.contains(v.node)) {
59             edgeDistance = v.cost;
60             newDistance = dist[u] + edgeDistance;
61
62             // If new distance is cheaper in cost
63             if (newDistance < dist[v.node])
64                 dist[v.node] = newDistance;
65
66             // Add the current node to the queue
67             pq.add(new Node(v.node, dist[v.node]));
68         }
69     }
70 }
```



```
110 // Class to represent a node in the graph
111 class Node implements Comparator<Node> {
112     public int node;
113     public int cost;
114
115     public Node()
116     {
117     }
118
119     public Node(int node, int cost)
120     {
121         this.node = node;
122         this.cost = cost;
123     }
124
125     @Override
126     public int compare(Node node1, Node node2)
127     {
128         if (node1.cost < node2.cost)
129             return -1;
130         if (node1.cost > node2.cost)
131             return 1;
132         return 0;
133     }
134 }
135 }
```

#### Output:

```
The shorted path from node :
0 to 0 is 0
0 to 1 is 8
0 to 2 is 6
0 to 3 is 5
0 to 4 is 3
```

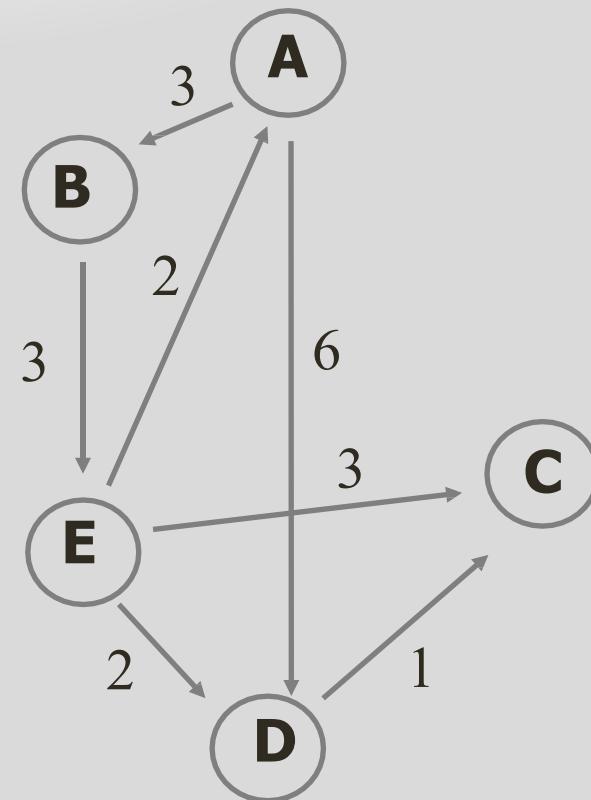
```
Dijkstra(Node start, Node goal)
{
    initialize all nodes' cost to infinity
    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal)
            {return} \\done!
        curr.visited = true
        foreach unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight)
            {
                PQ.enqueue(n) || update n's position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight
            }
        }
    }
}
```

- what path will Dijkstra's find from A to C?

A) A B E C

B) A D C

C) A B E D C



# Heaps

## CSC220|Computer Programming 2

Last Time...

# Deletion

- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

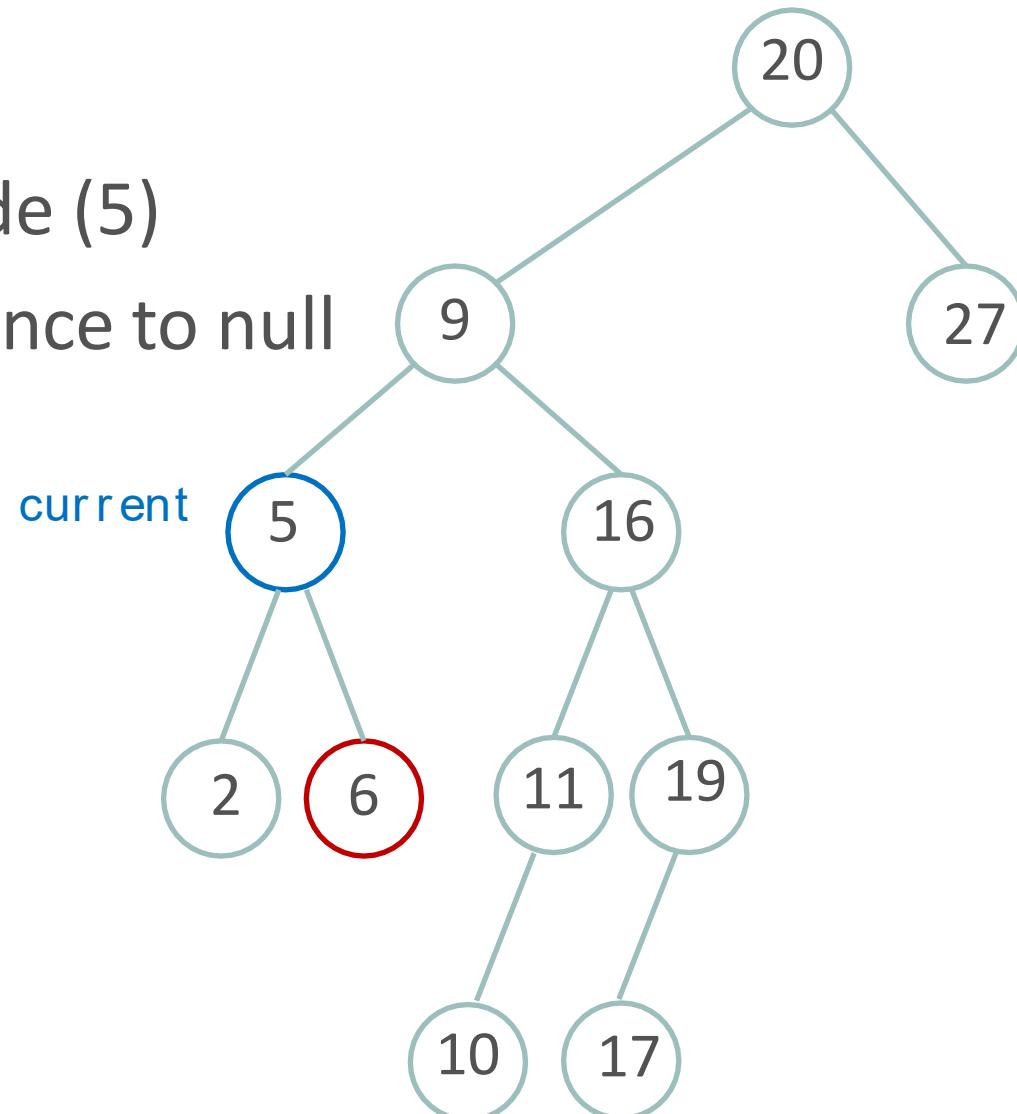
- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list
- There are three different cases:
  - 1) Deleting a leaf node
  - 2) Deleting a node with one child subtree
  - 3) Deleting a node with two children subtrees

- Since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list
- There are three different cases:
  - 1) Deleting a leaf node
  - 2) Deleting a node with one child subtree
  - 3) Deleting a node with two children subtrees
- First step of deletion is to find the node to delete
  - Just a regular BST search
  - BUT, stop at the *parent* of the node to be deleted

# case 1: Deleting a leaf node

## DELETE NODE 6

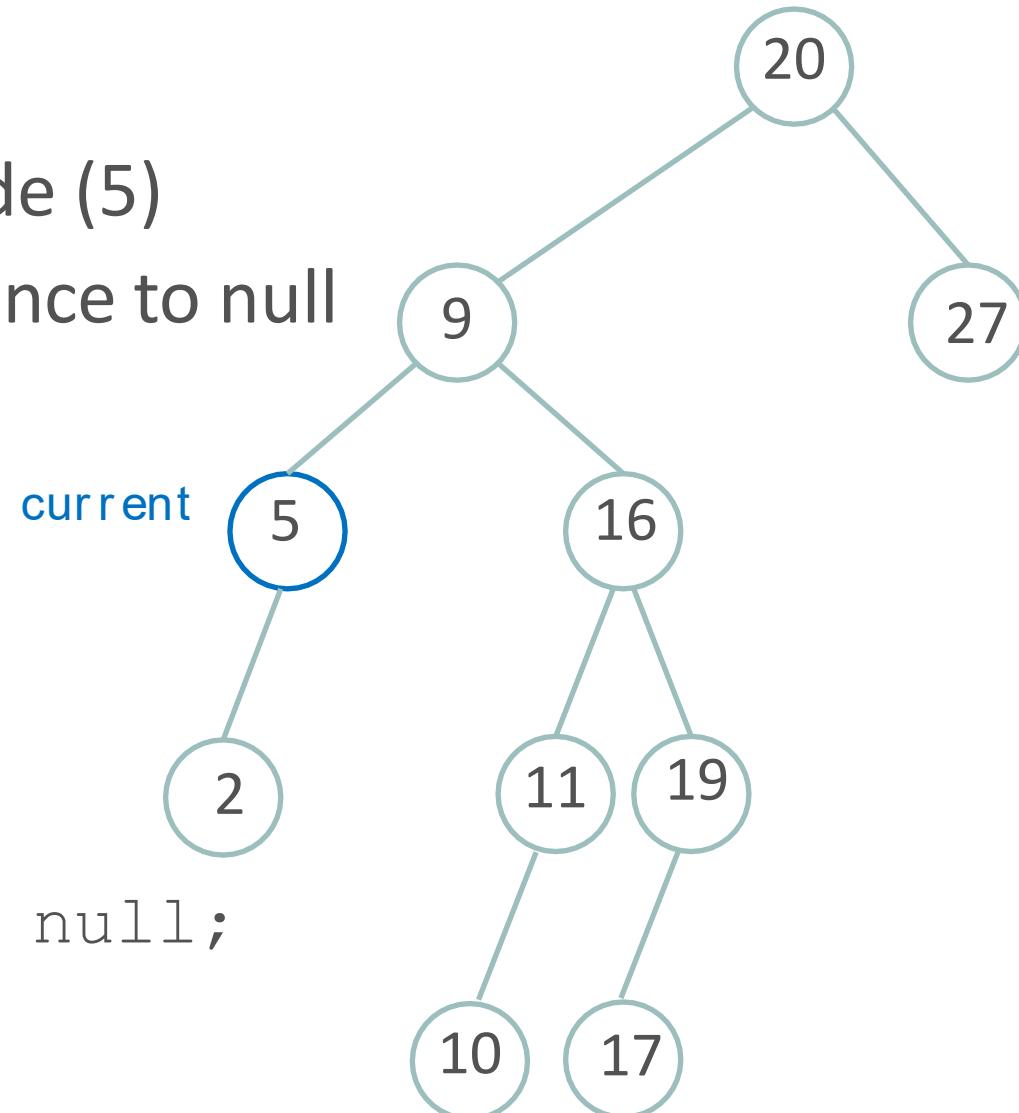
- Stop at parent node (5)
- Set parent's reference to null



# case 1: Deleting a leaf node

## DELETE NODE 6

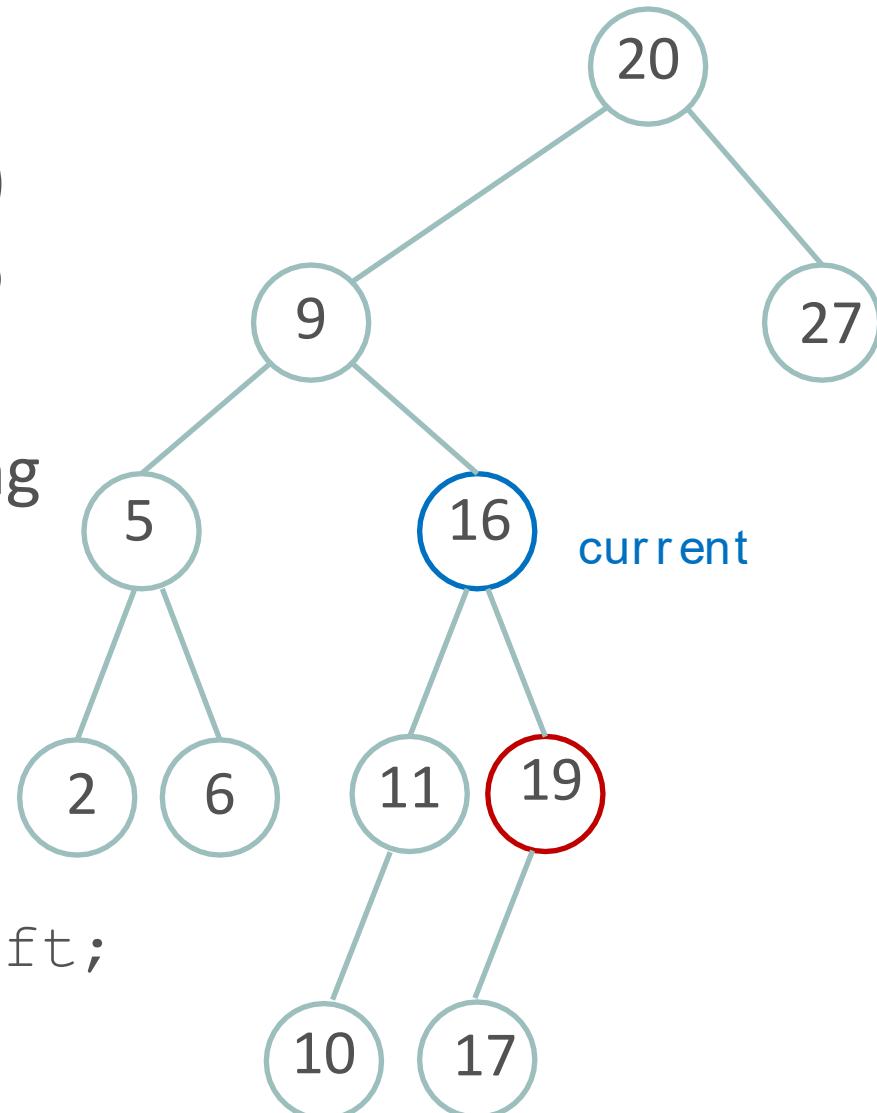
- Stop at parent node (5)
- Set parent's reference to null



# case 2: Delete node with 1 child

## DELETE NODE 19

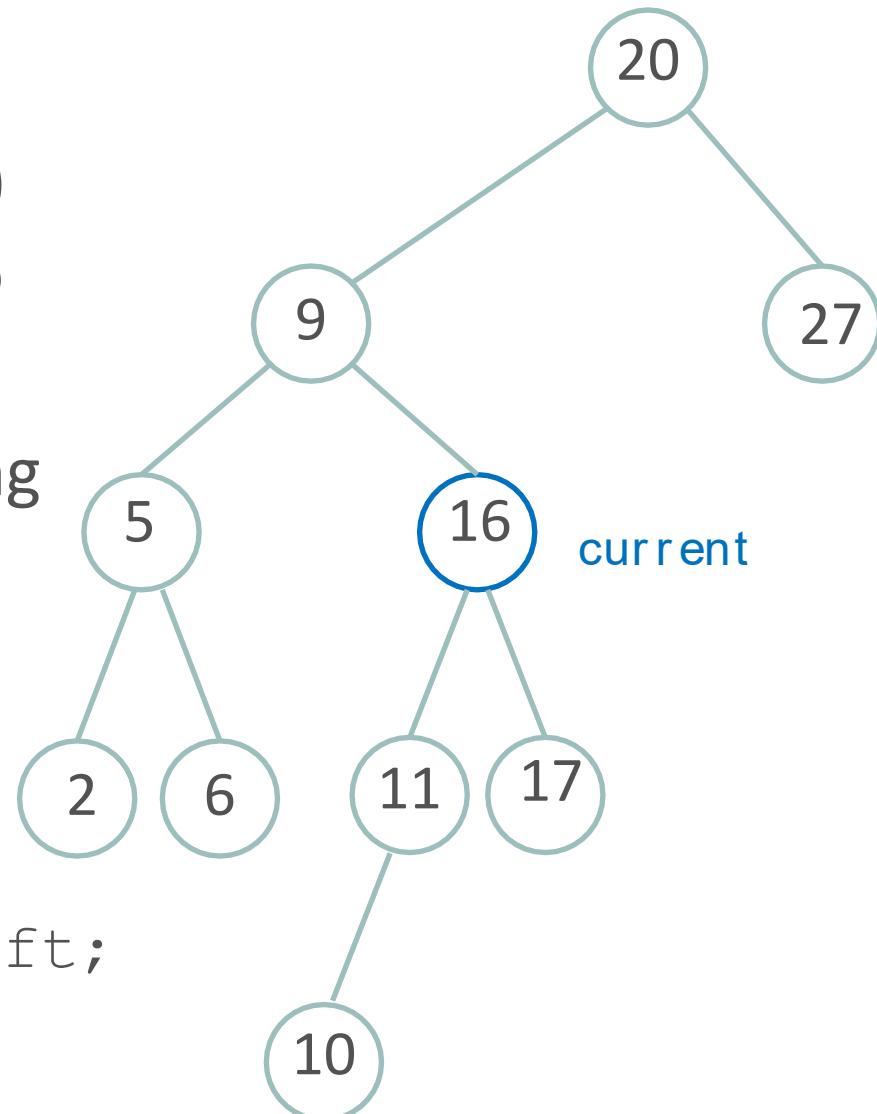
- Stop at parent node (16)
- Set parent's reference to node's child
- Multiple cases depending on which side the child and grandchild are on!



# case 2: Delete node with 1 child

## DELETE NODE 19

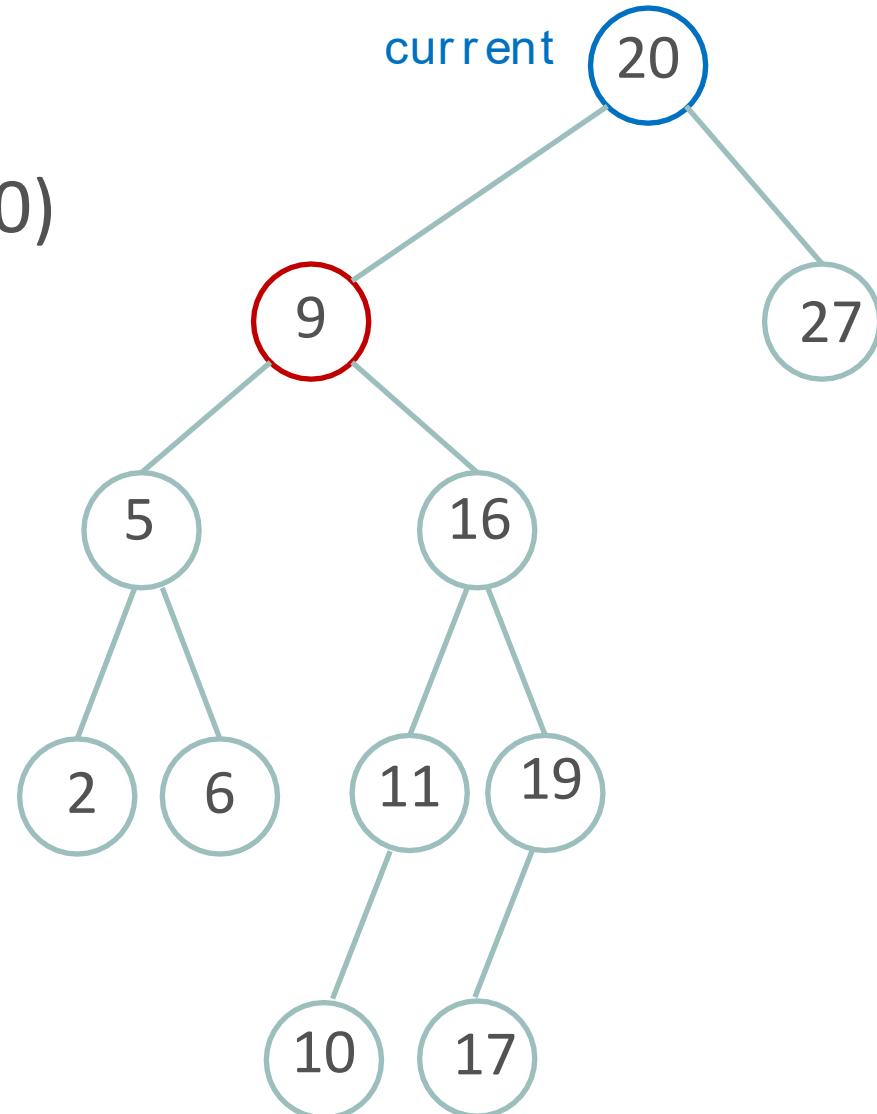
- Stop at parent node (16)
- Set parent's reference to node's child
- Multiple cases depending on which side the child and grandchild are on!



# case 3: Delete node with 2 children

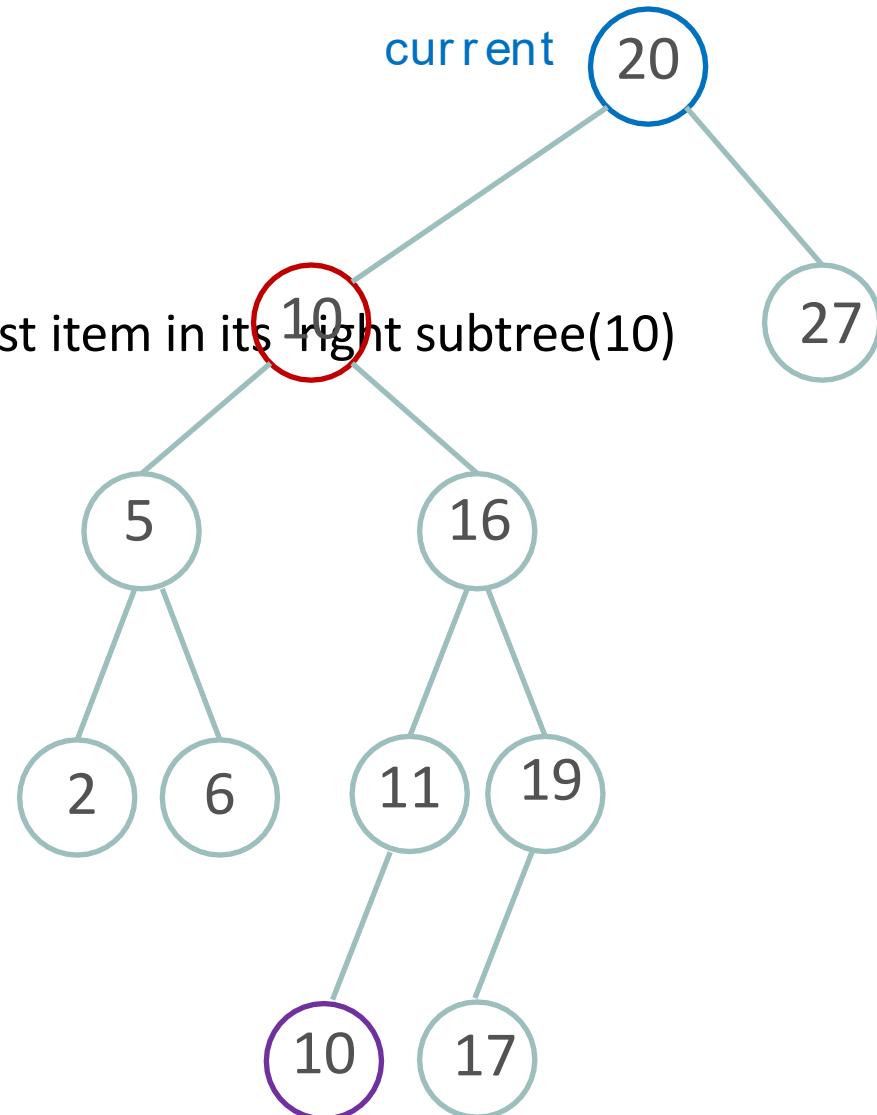
## DELETE NODE 9

- Stop at parent node (20)
- Replace the node with smallest item in its right subtree(10)



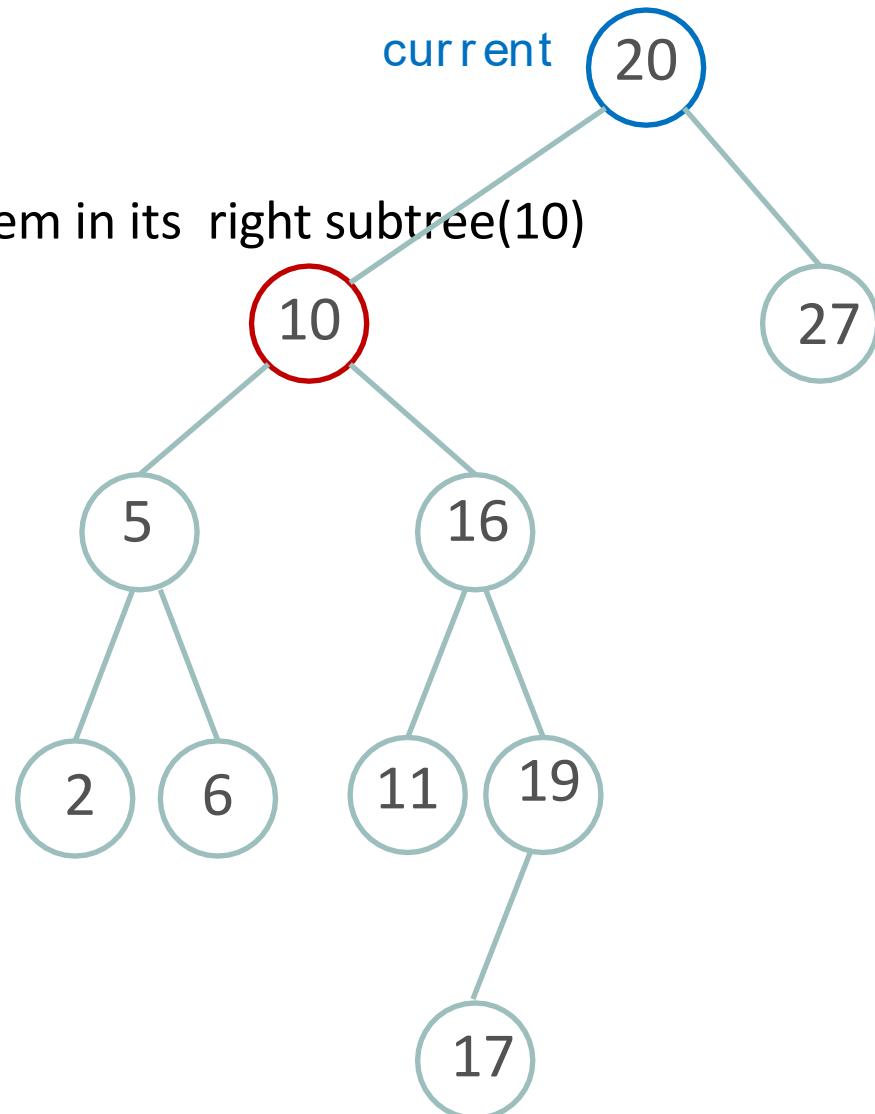
# case 3: Delete node with 2 children

- DELETE NODE 9
- Stop at parent node (20)
- Replace the node with smallest item in its right subtree(10)
- Perform a delete on
  - on successor (10)



# case 3: Delete node with 2 children

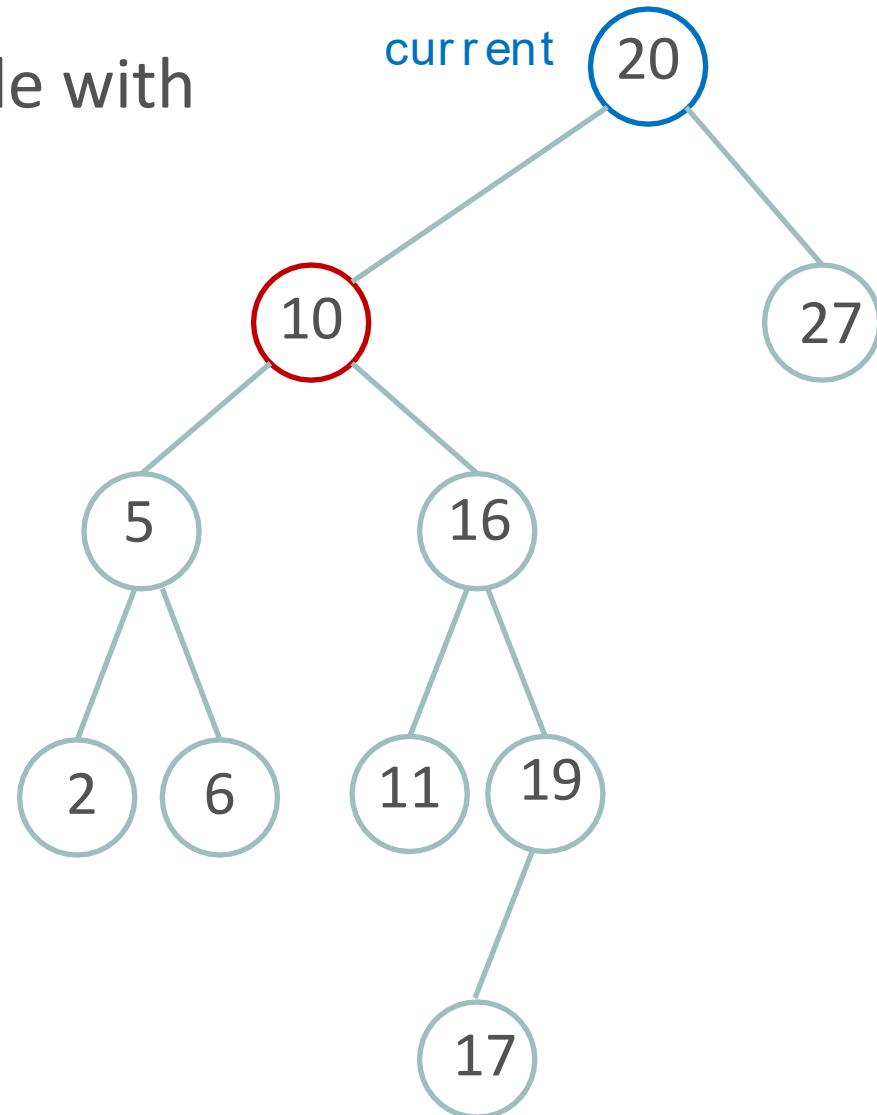
- DELETE NODE 9
- Stop at parent node (20)
- Replace the node with smallest item in its right subtree(10)
- Perform a delete on
  - on *successor* (10)



# case 3: Delete node with 2 children

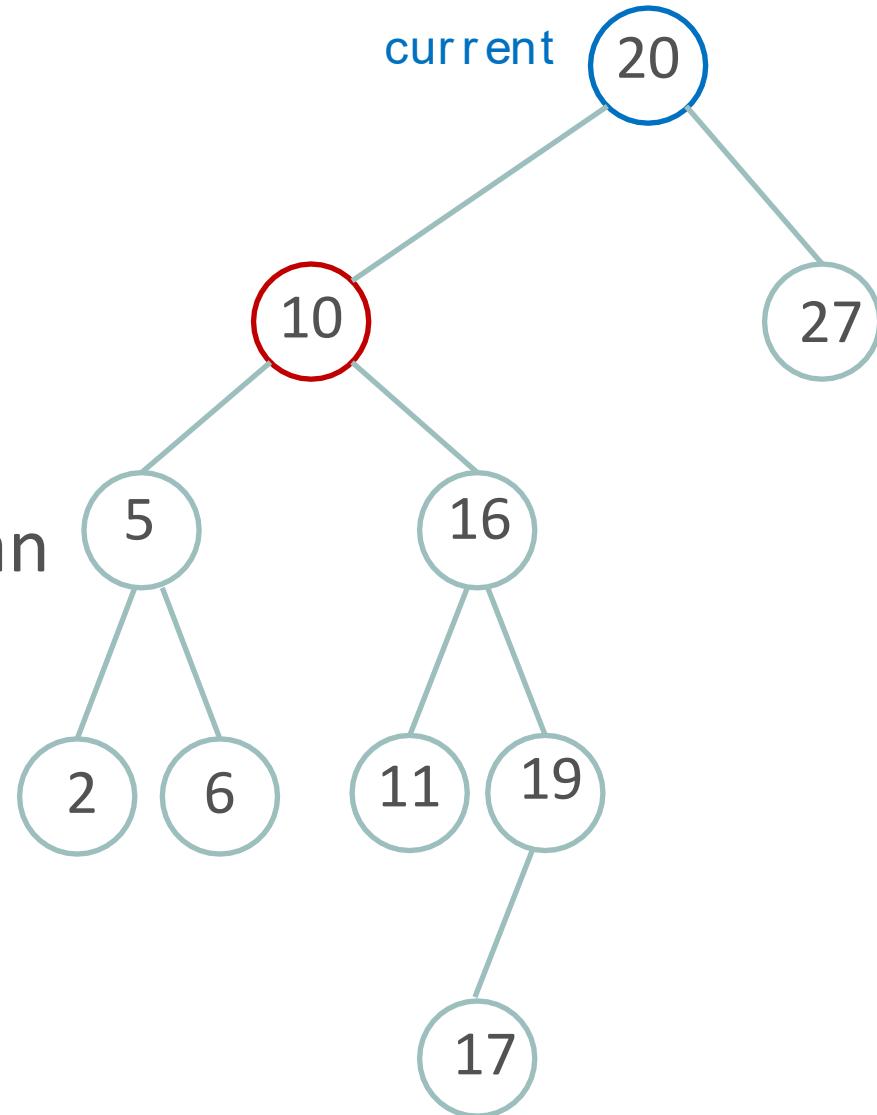
Replacing the deleted node with  
the smallest child in right  
subtree guarantees  
the BST structure...

Why?



# case 3: Delete node with 2 children

The smallest item in the right subtree is greater than any item in the left subtree, *and* smaller than any item in the new right subtree.



# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:
- Cost of:
  - Case 1 (delete leaf):
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete: **O(log N)**
- Cost of:
  - Case 1 (delete leaf):
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete: **O(log N)**
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null: O(1)**
  - Case 2 (delete node with 1 child):
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:  $O(\log N)$
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null:  $O(1)$**
  - Case 2 (delete node with 1 child):  
**bypass a reference:  $O(1)$**
  - Case 3 (delete node with 2 children):

# Deletion performance

- What is the cost of deleting a node from a BST?
- First, find the node we want to delete:  $O(\log N)$
- Cost of:
  - Case 1 (delete leaf):  
**set a single reference to null:  $O(1)$**
  - Case 2 (delete node with 1 child):  
**bypass a reference:  $O(1)$**
  - Case 3 (delete node with 2 children):  
**Find the successor:  $O(\log N)$**   
**delete the duplicate successor:  $O(1)$**

# Other useful properties

- How can we find the smallest node?

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?

# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree

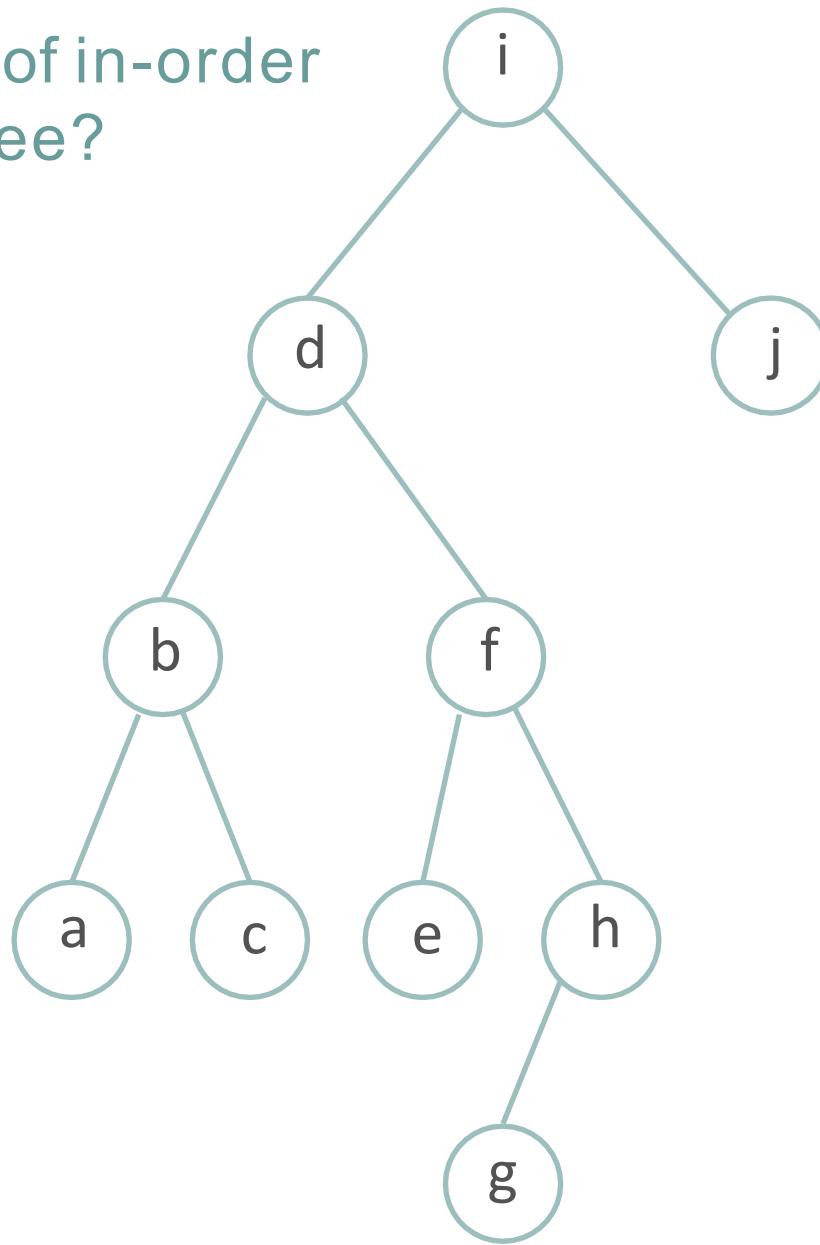
# Other useful properties

- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree
- How can we print nodes in ascending order?

# Other useful properties

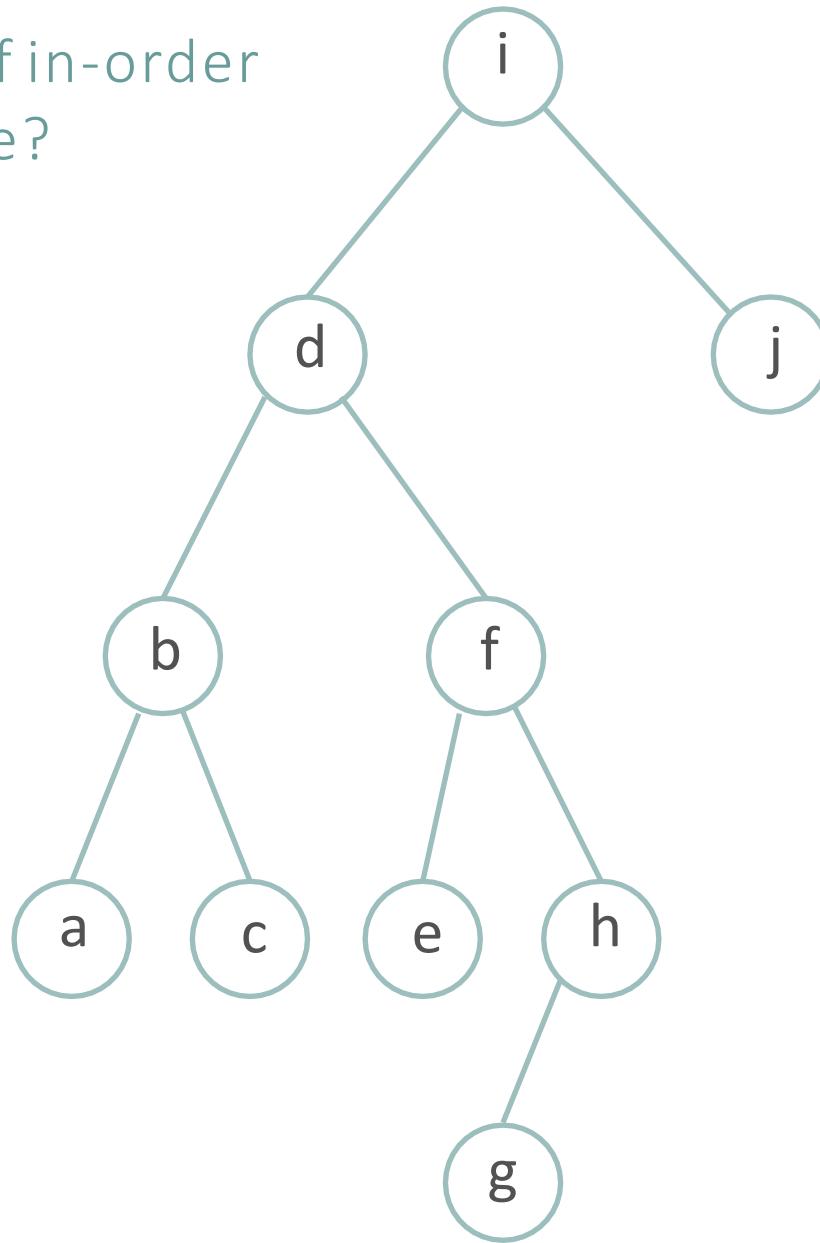
- How can we find the smallest node?
  - Start at the root, go as far left as possible
  - $O(\log N)$  on a balanced tree
- How can we find the largest node?
  - Start at the root, go as far right as possible
  - $O(\log N)$  on a balanced tree
- How can we print nodes in ascending order?
  - In-order traversal

What is the result of in-order traversal of this tree?



a b c d e f g h i j

What is the result of in-order traversal of this tree?



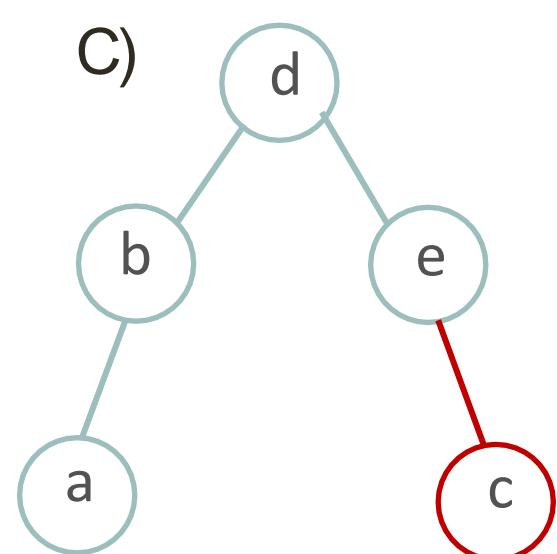
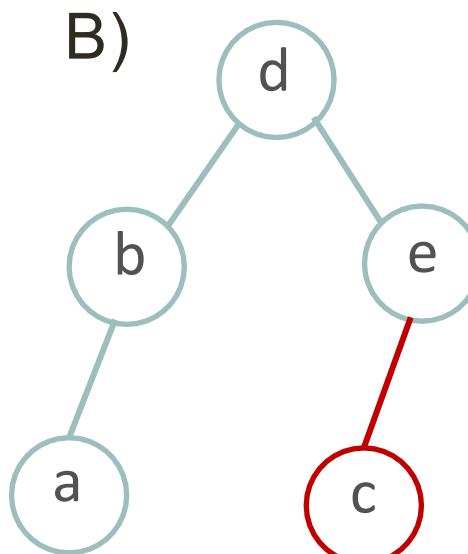
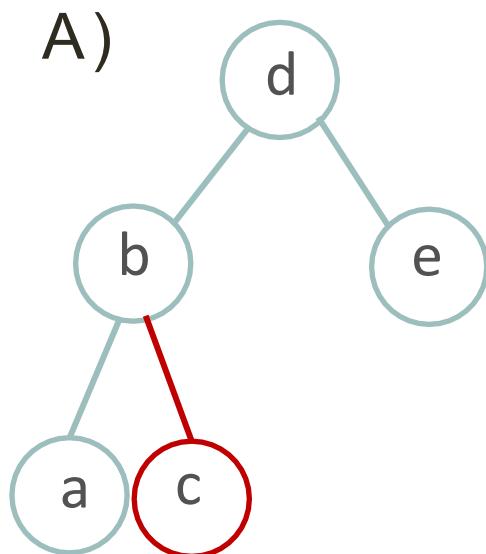
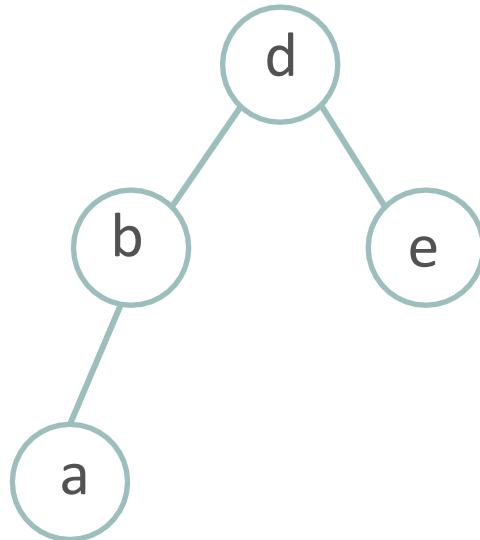
Alphabetical order  
Or  
Lexicographical order



a b c d e f g h i j

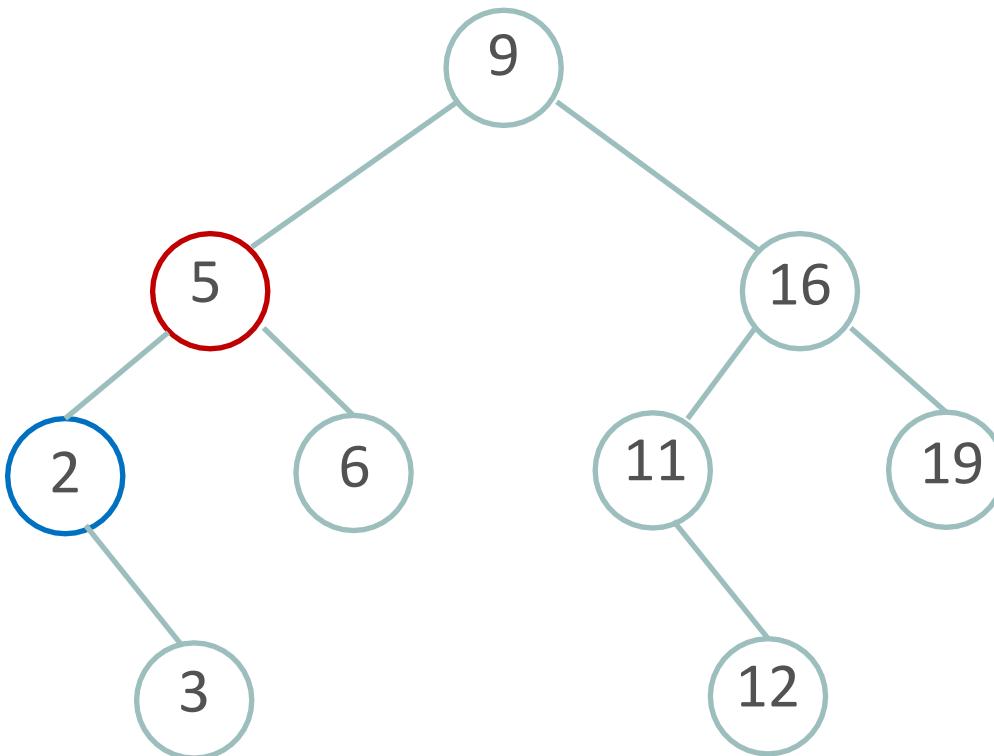
# Review

Which of the following trees is the result of adding c to this bst?



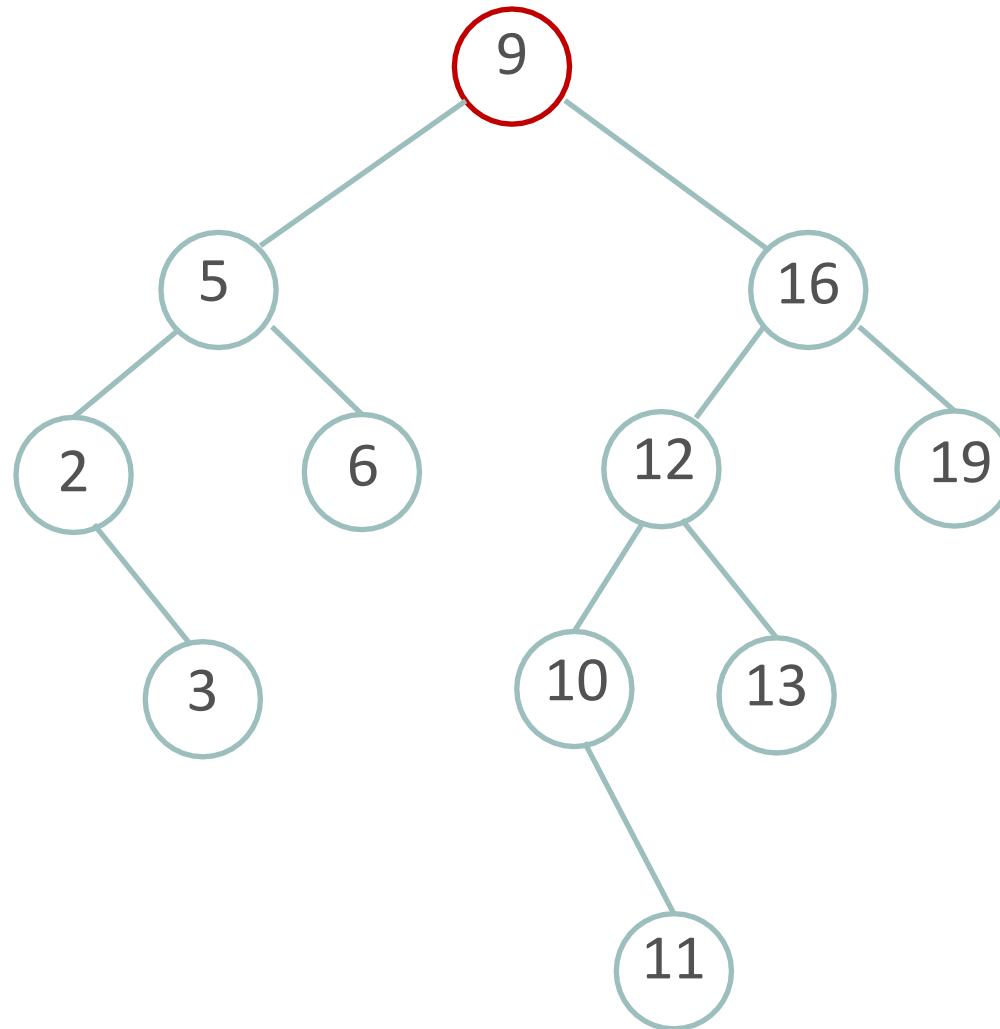
What will 5's left child be after deleting 2?

- A) 3
- B) 6
- C) null



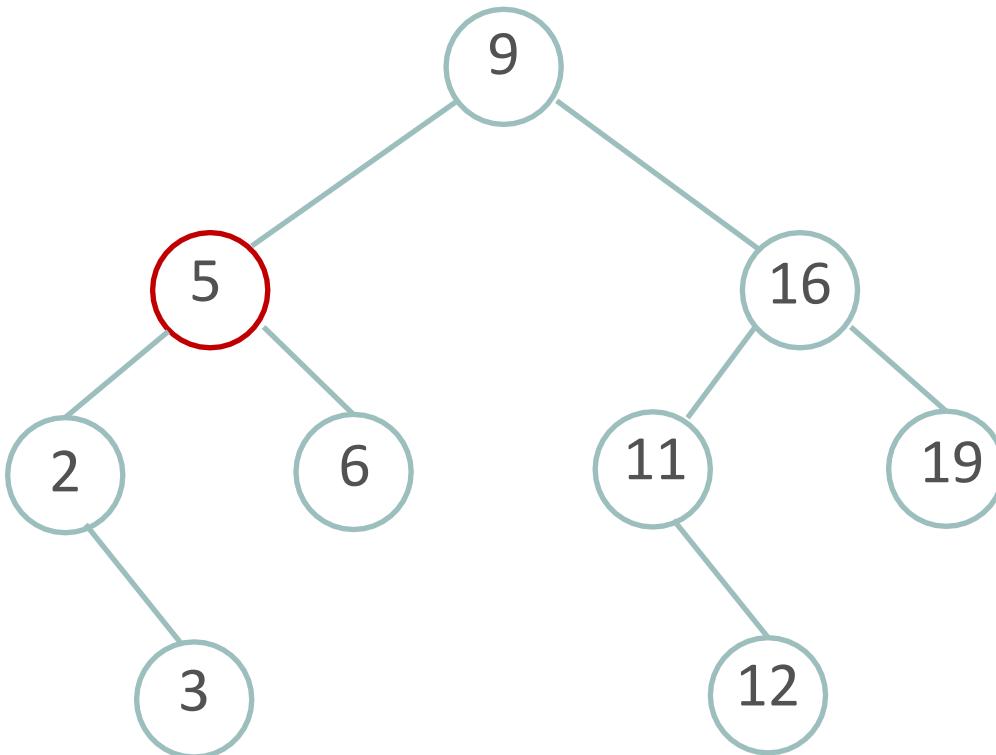
If we delete 9, what gets replaced?

- A) 6
- B) 10
- C) 13
- D) 19



# What happens if we delete 5?

- A) 2
- B) 3
- C) 6
- D) 12



Let's build some trees

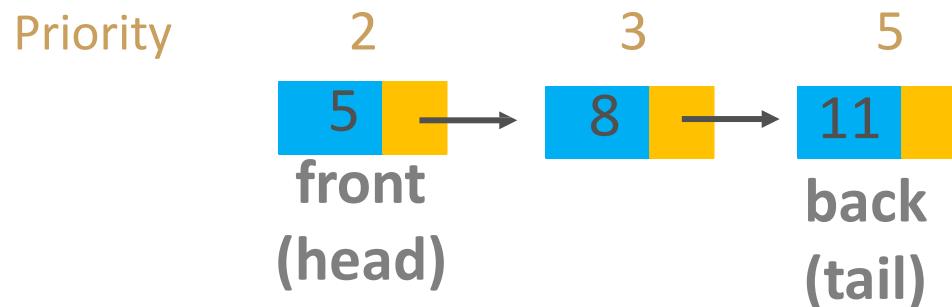
Today...

# Priority Queue

- A priority queue is a data structure in which access is limited to the minimum item in the set
  - add
  - findMin
  - deleteMin
- Add location is unspecified, so long as the above is always enforced
- What are our options for implementing this?

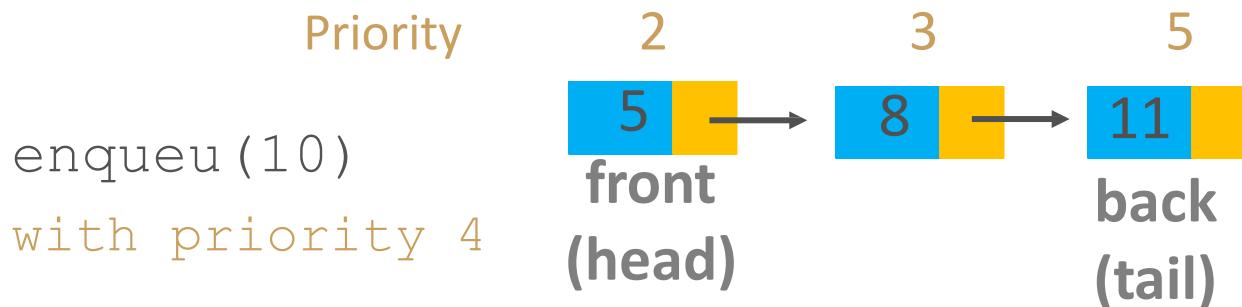
# Priority queue as a linked list...

- Always add items in correct, sorted spot



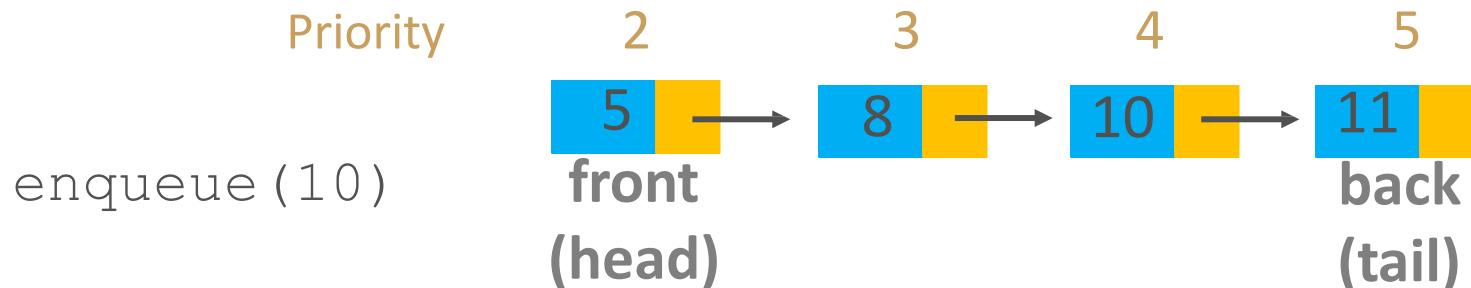
# Priority queue as a linked list...

- Always add items in correct, sorted spot



# Priority queue as a linked list...

- Always add items in correct, sorted spot



- dequeue will return smallest item  $\circ(1)$

- Option 1: a linked list
  - add:
  - findMin:
  - deleteMin: (including finding)
- Option 2: a sorted linked list
  - add:
  - findMin:
  - deleteMin:

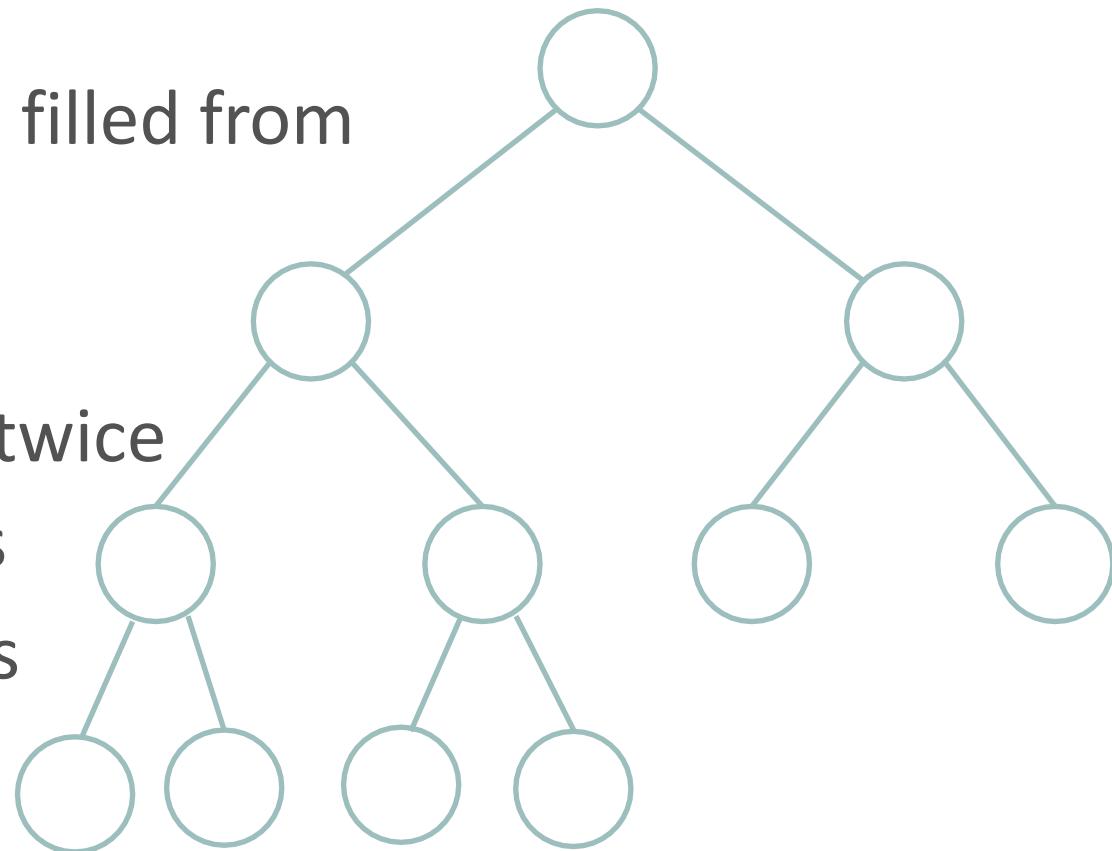
- Option 1: a linked list
  - add:  $O(1)$
  - findMin:  $O(N)$
  - deleteMin:  $O(N)$  (including finding)
- Option 2: a sorted linked list
  - add:  $O(N)$
  - findMin:  $O(1)$
  - deleteMin:  $O(1)$

- Option 1: a linked list
  - add:  $O(1)$
  - findMin:  $O(N)$
  - deleteMin:  $O(N)$  (including finding)
- Option 2: a sorted linked list
  - add:  $O(N)$
  - findMin:  $O(1)$
  - deleteMin:  $O(1)$
- Option 3: a binary heap
  - add:  $O(1)$
  - findMin:  $O(1)$
  - deleteMin:  $O(\log N)$

# Complete Trees

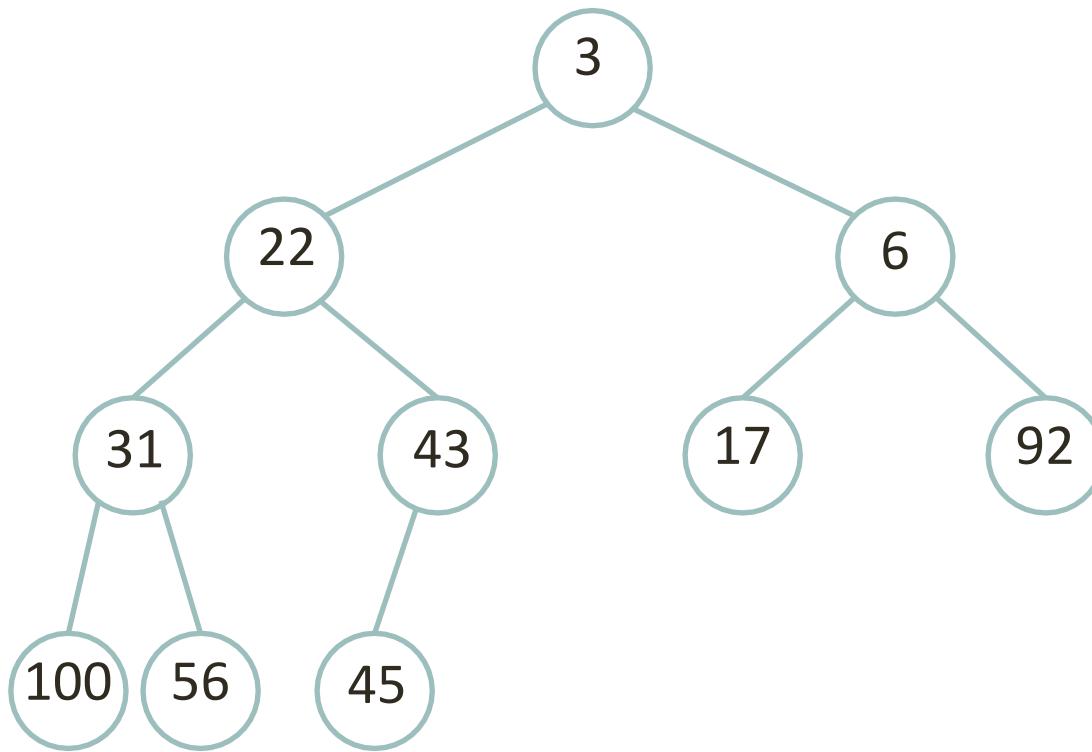
- A **complete binary tree** has its levels completely filled, with the possible exception of the bottom level

- Bottom level is filled from left to right
- Each level has twice as many nodes as the previous level



# Binary Heaps

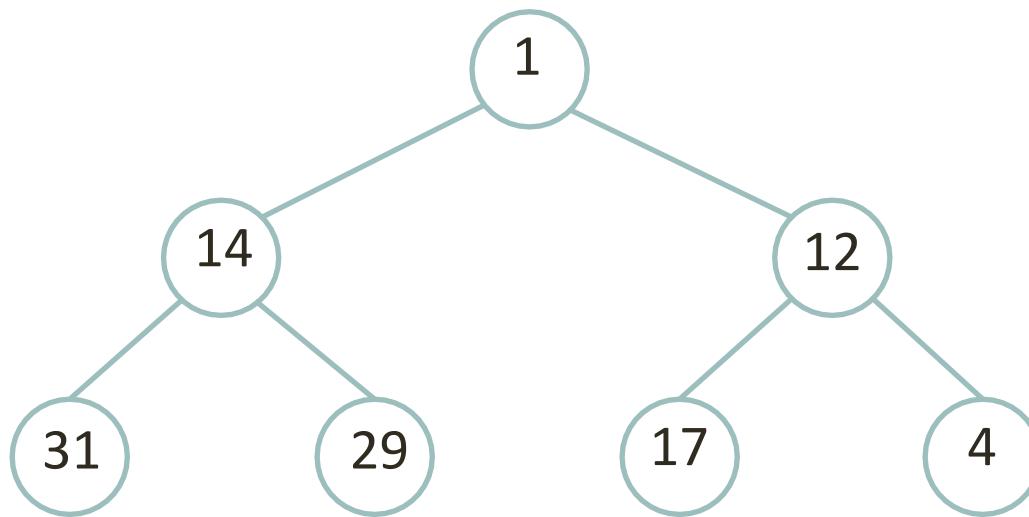
- A **binary heap** is a binary tree with two special properties
  - *Structure*: it is a complete tree
  - *Order*: the data in any node is less than or equal to the data of its children
- This is also called a **min-heap**
  - A **max-heap** would have the opposite property



- Order of children does not matter, only that they are greater than their parent

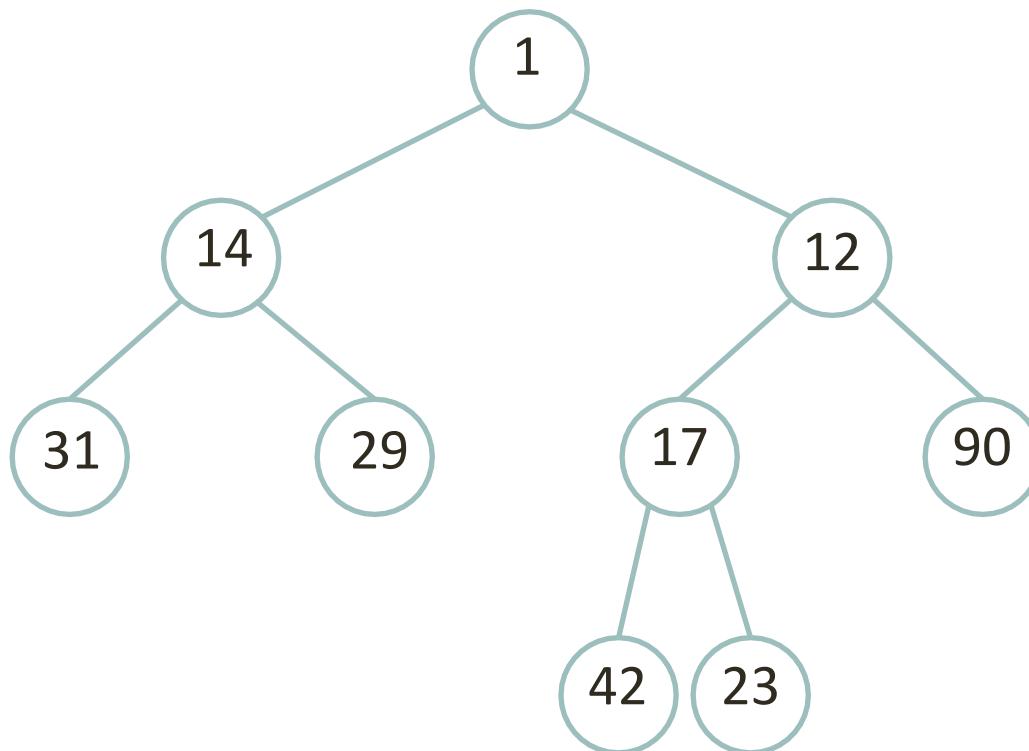
Is this a min-heap?

- A) yes
- B) no



Is this a min-heap?

- A) yes
- B) no



# Recap

- Heap versus priority queue
  - Priority queue: a description of a set of operations
  - Heap: one possible efficient way of implementing those operations

# Adding to a heap

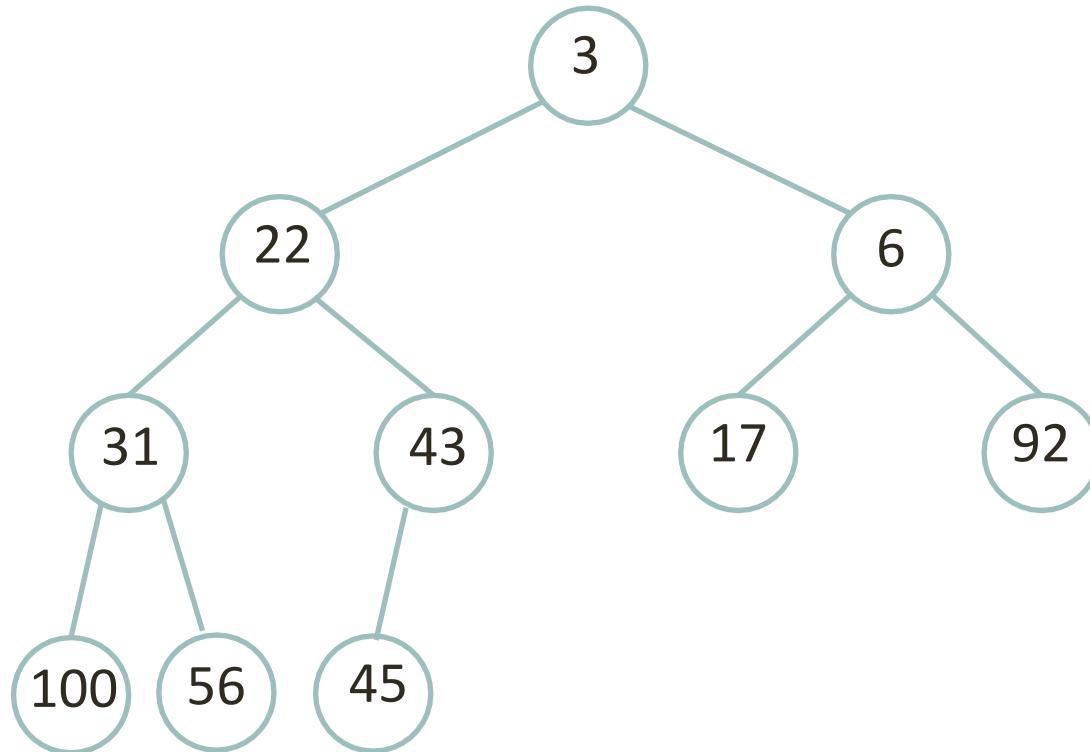
- We must be careful to maintain the two properties when adding to a heap
  - Structure
  - Order
- Deal with the structure property first...
  - where can the new item go to maintain a complete tree?
- Then, *percolate* the item upward until the order property is restored
  - Swap upwards until > parent

# Percolate (bubble)

- Moving a heap element upward (or downward) until the heap's state satisfies the required vertical ordering.

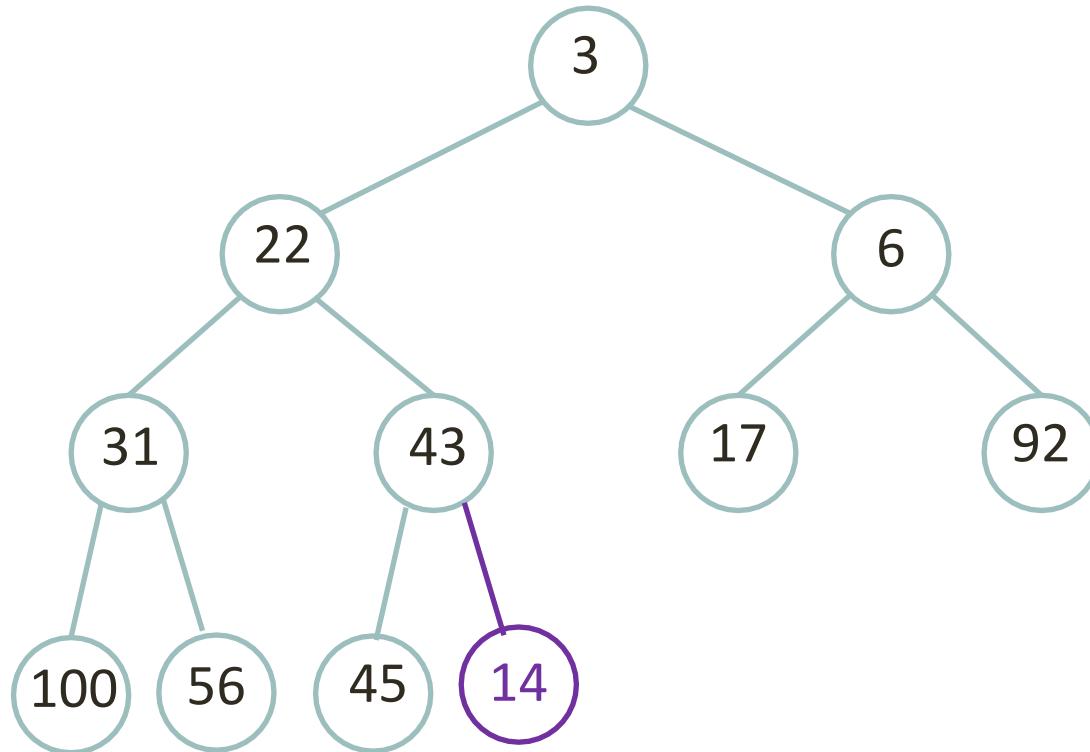
# Adding 14

Put it at the end of the tree



# Adding 14

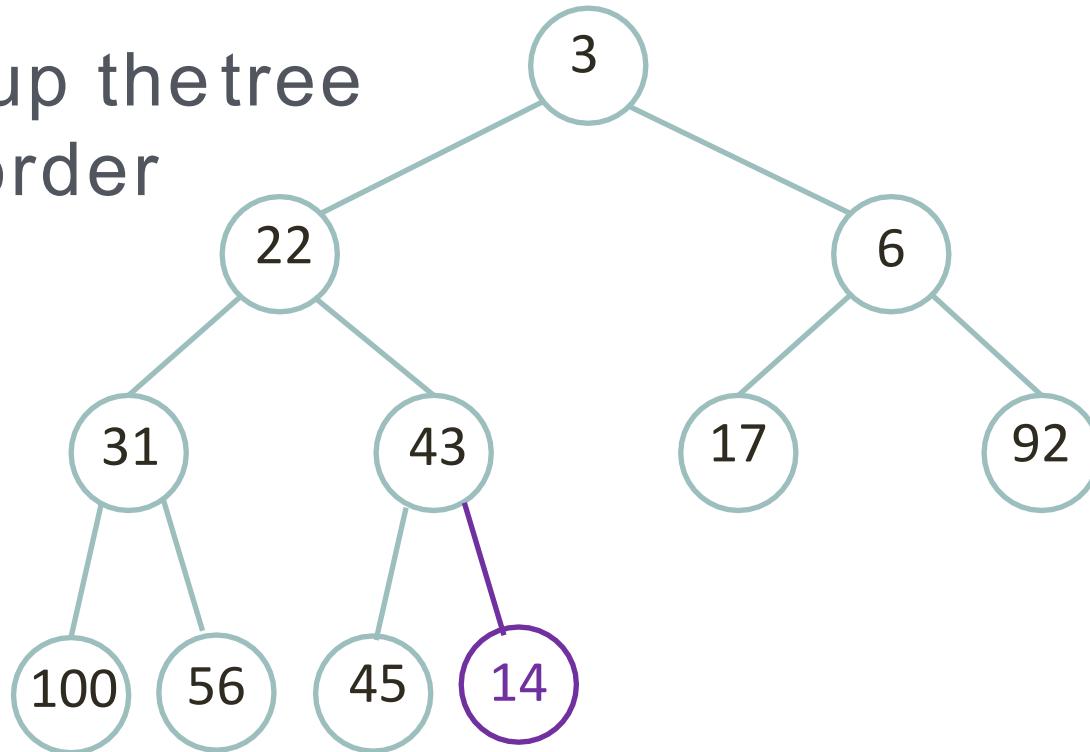
Put it at the end of the tree



# Adding 14

Put it at the end of the tree

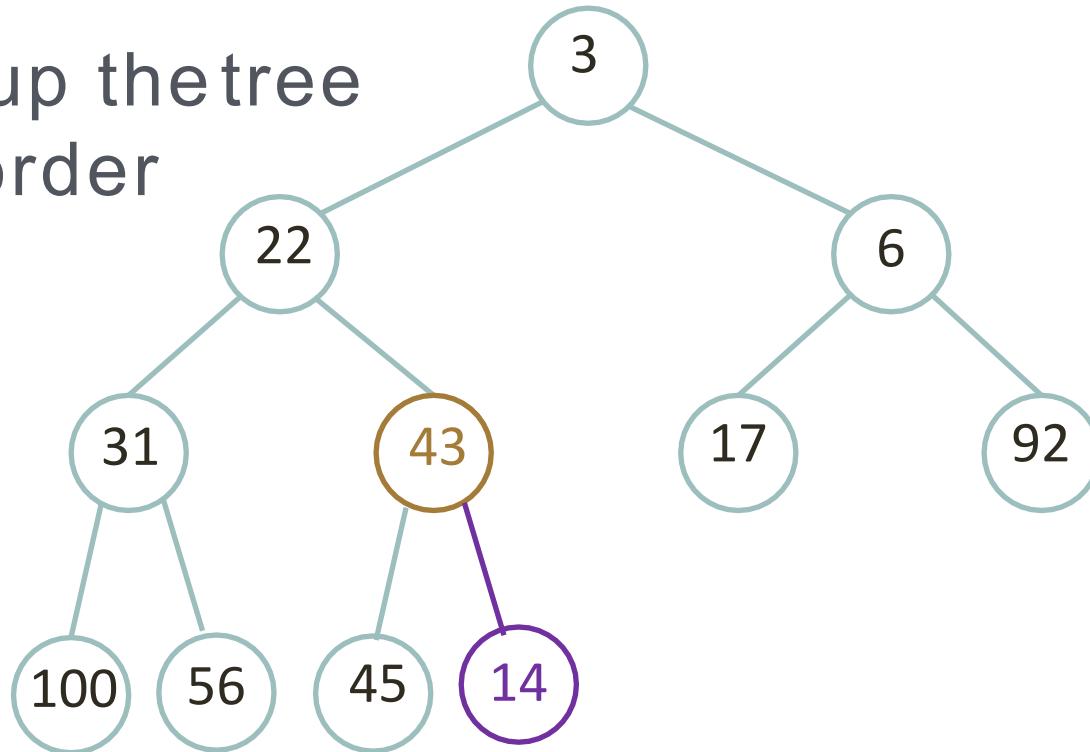
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

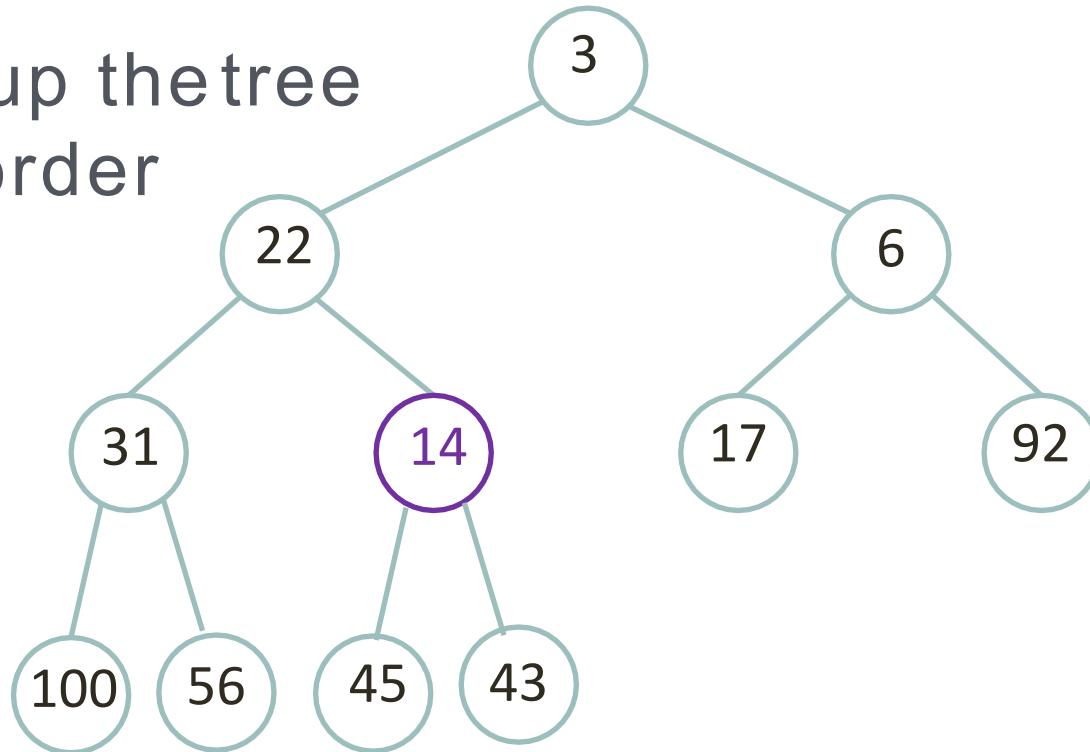
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

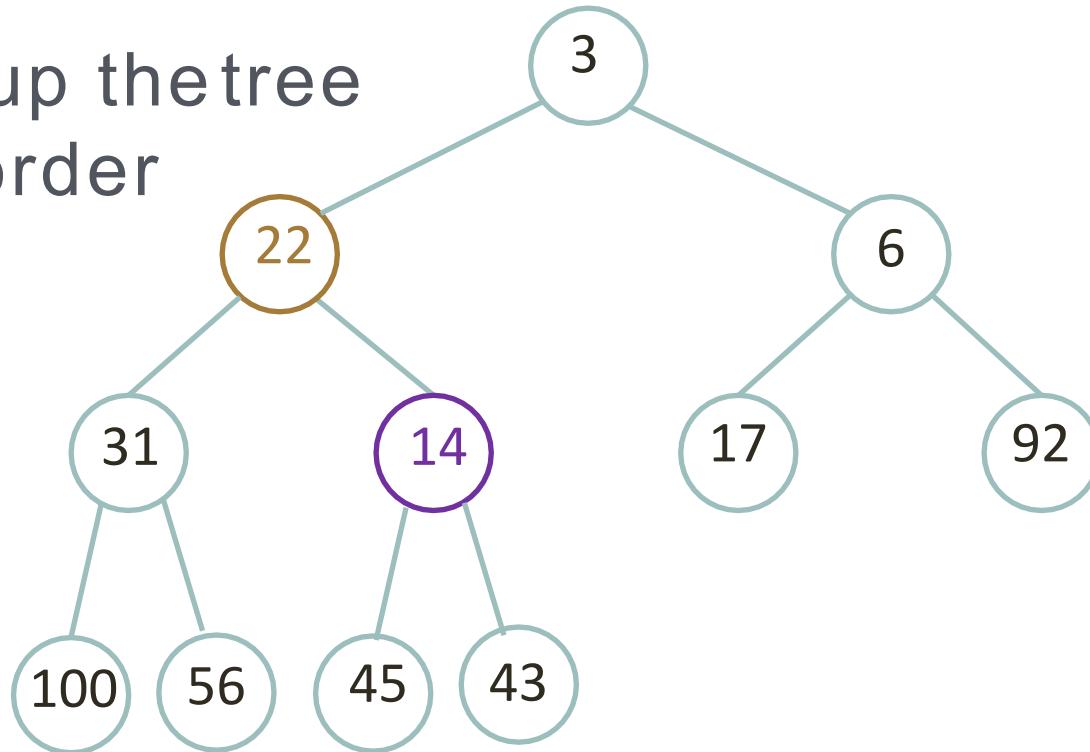
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

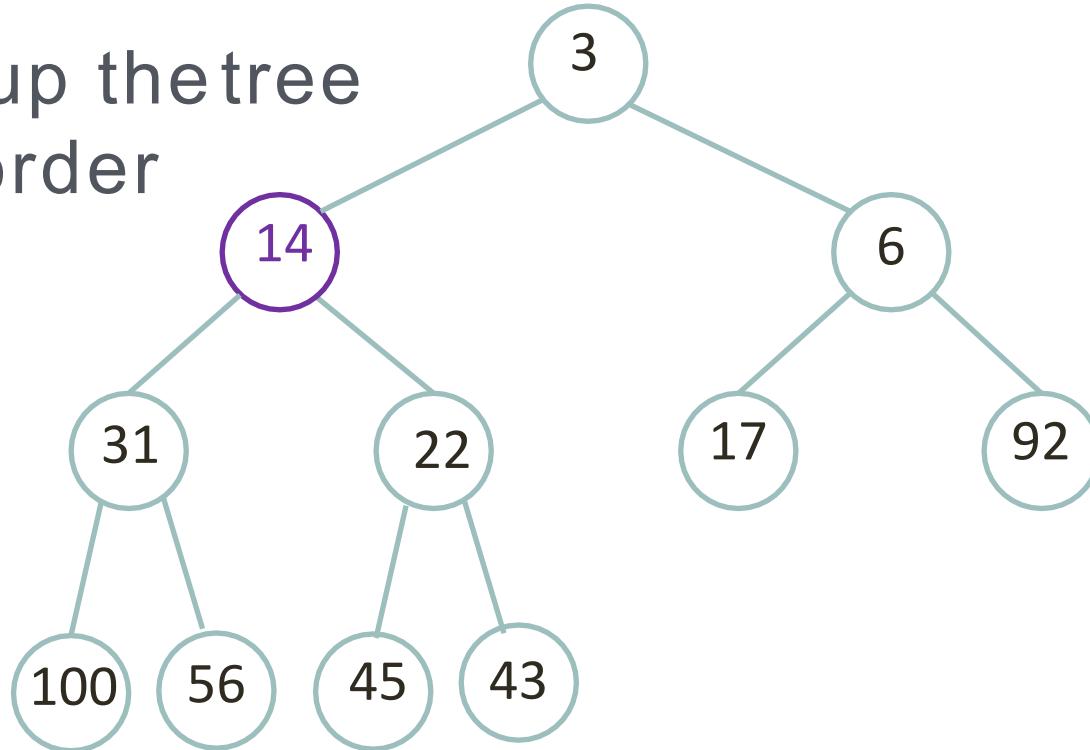
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

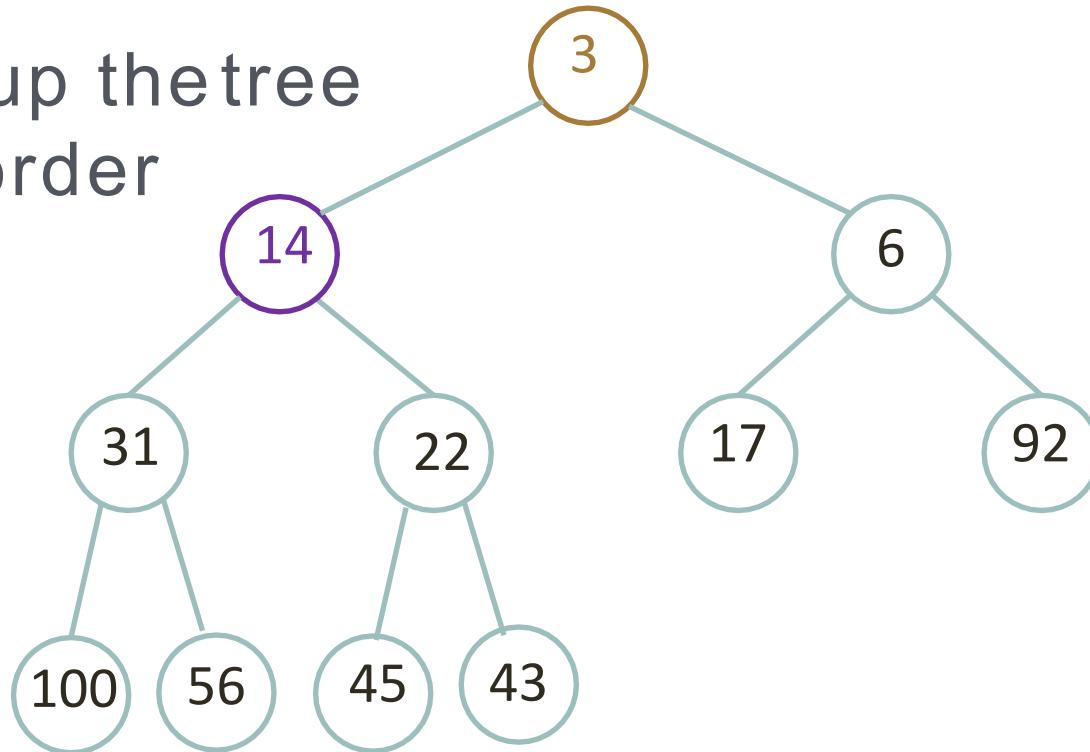
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

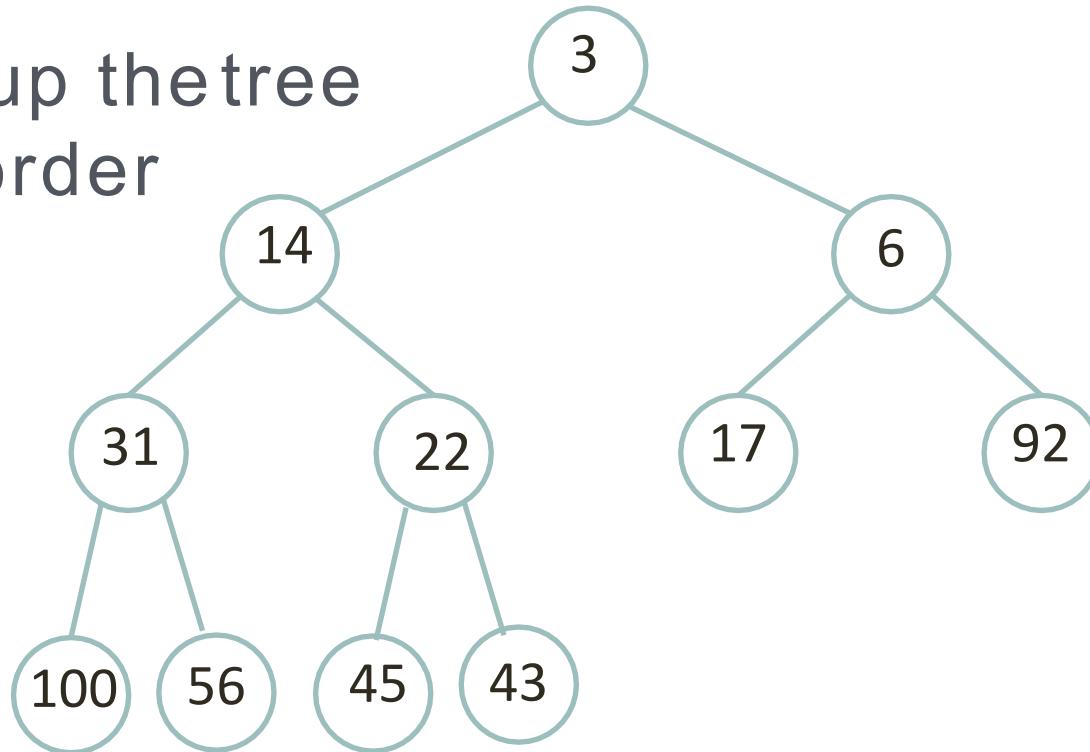
Percolate up the tree  
to fix the order



# Adding 14

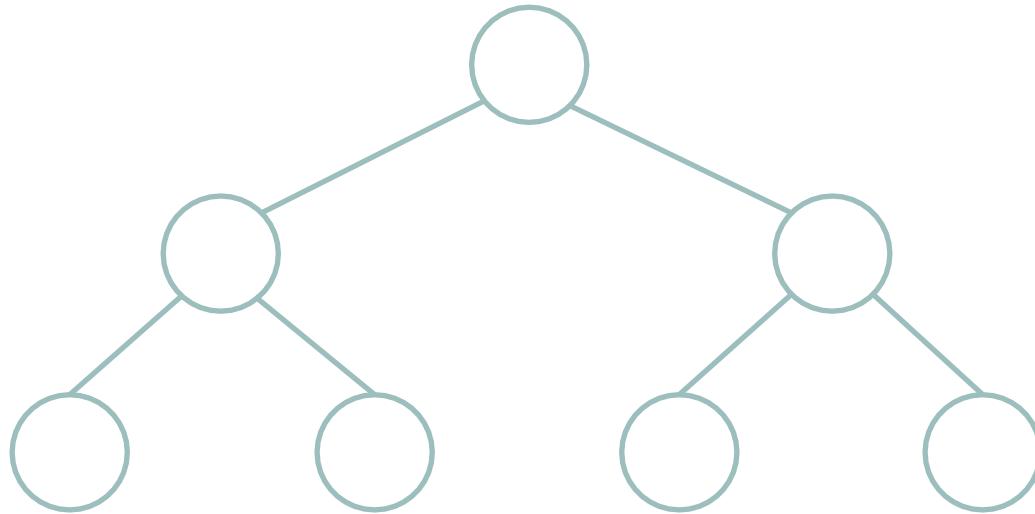
Put it at the end of the tree

Percolate up the tree  
to fix the order



# Cost of add

- Percolate up until smaller than all nodes below it...
- How many nodes are there on each level (in terms of  $N$ )?



- If the new item is the smallest in the set, cost is  $O(\log N)$ 
  - Must percolate up every level to the root
  - Complete trees have  $\log N$  levels
    - *Is this the worst, average, or best case?*
- It has been shown that on average, 2.6 comparisons are needed for any  $N$ 
  - Thus, the adding action terminates early, and average cost is  $O(1)$

# Pseudo code...

Insert the new element value as a new rightmost leaf node

Current node = the new leaf node

While (current node's value is not greater than its parent) {

    swap current node's value with parent's value

    current node = parent node

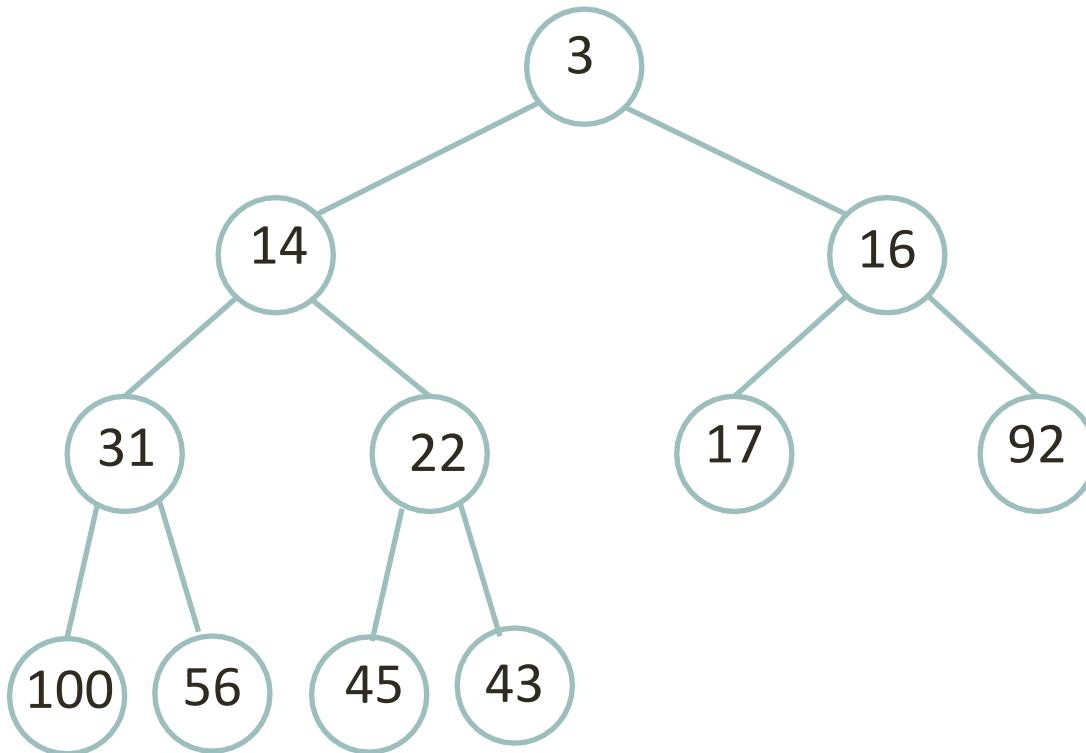
}

# Remove

# Reverse of add!

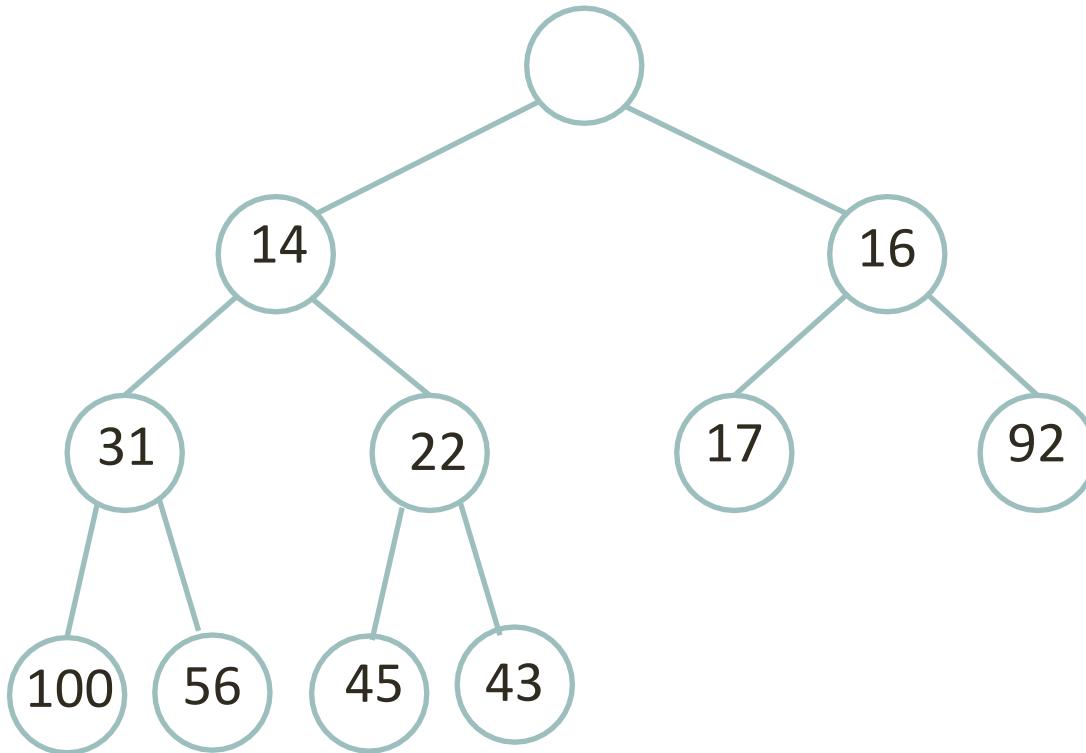
# Let's remove the smallest item

Take out 3



# Let's remove the smallest item

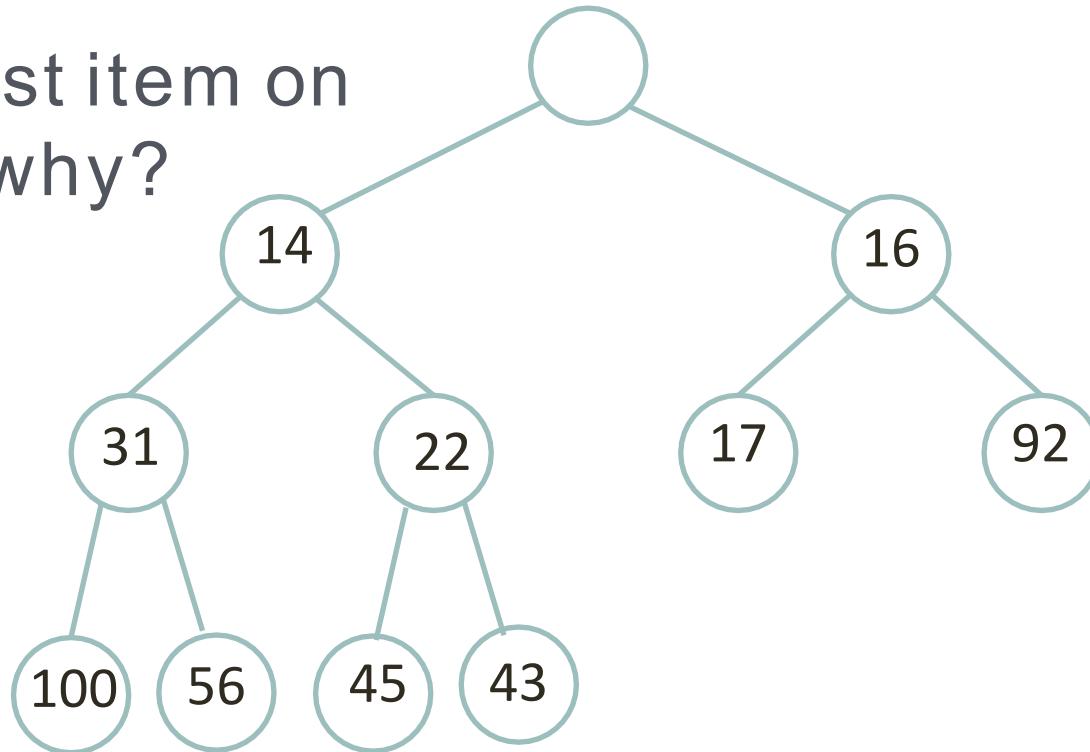
Take out 3



# Let's remove the smallest item

Take out 3

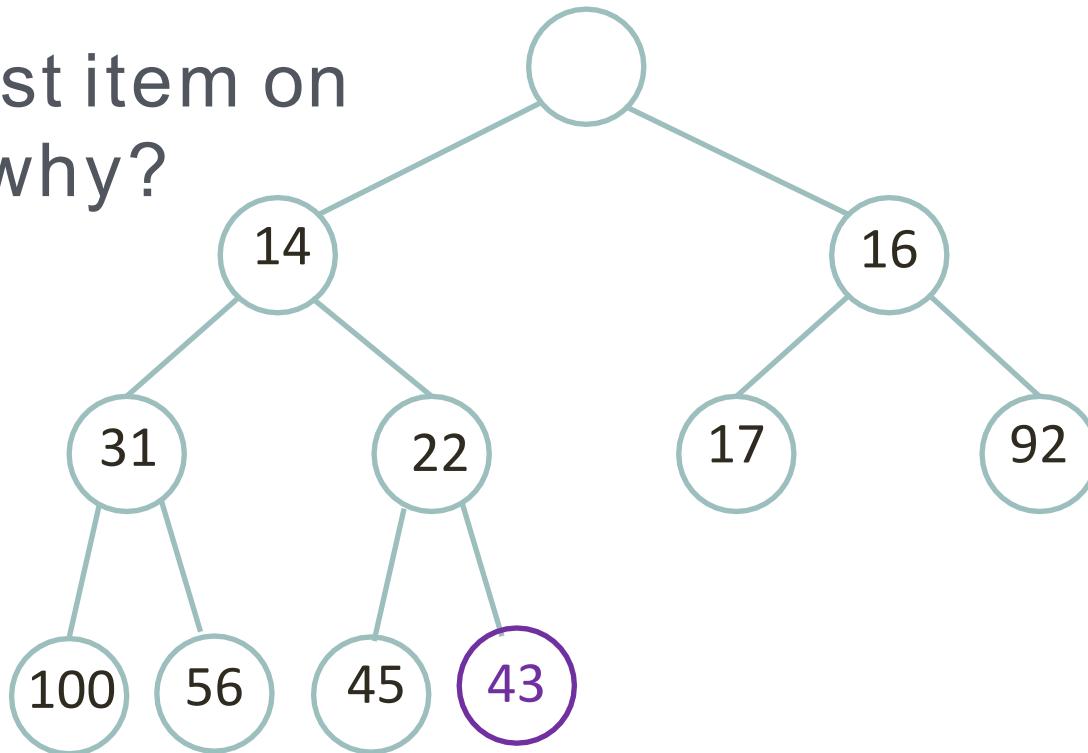
Fill with last item on last level. why?



# Let's remove the smallest item

Take out 3

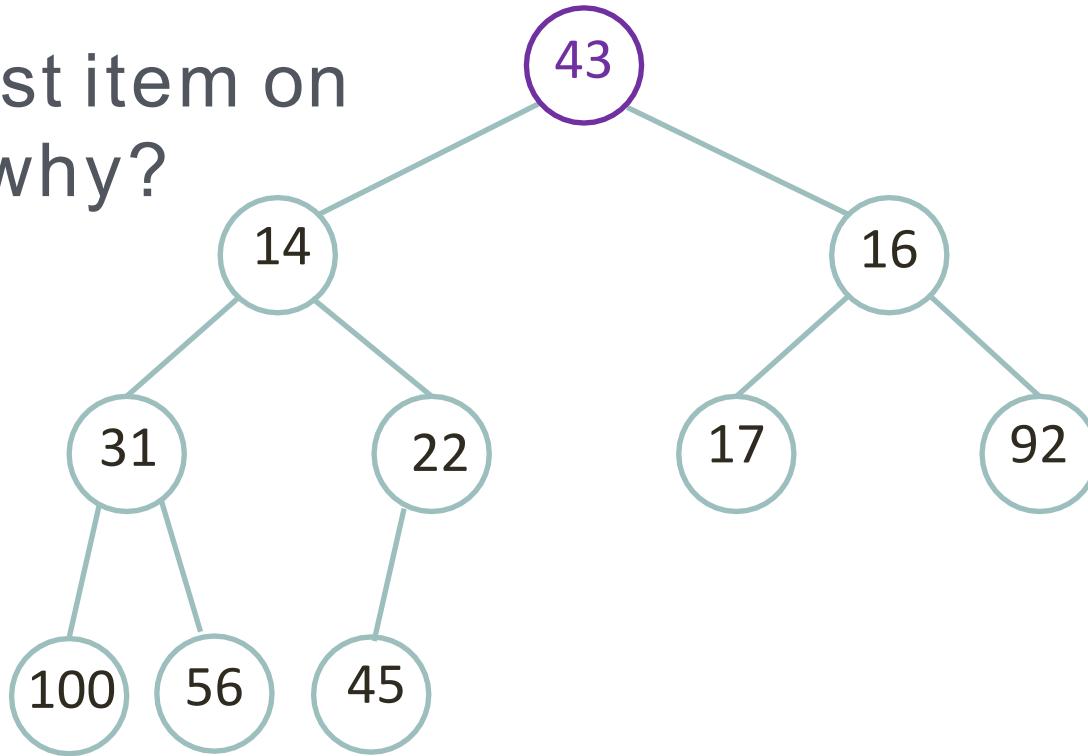
Fill with last item on last level. why?



# Let's remove the smallest item

Take out 3

Fill with last item on last level. why?

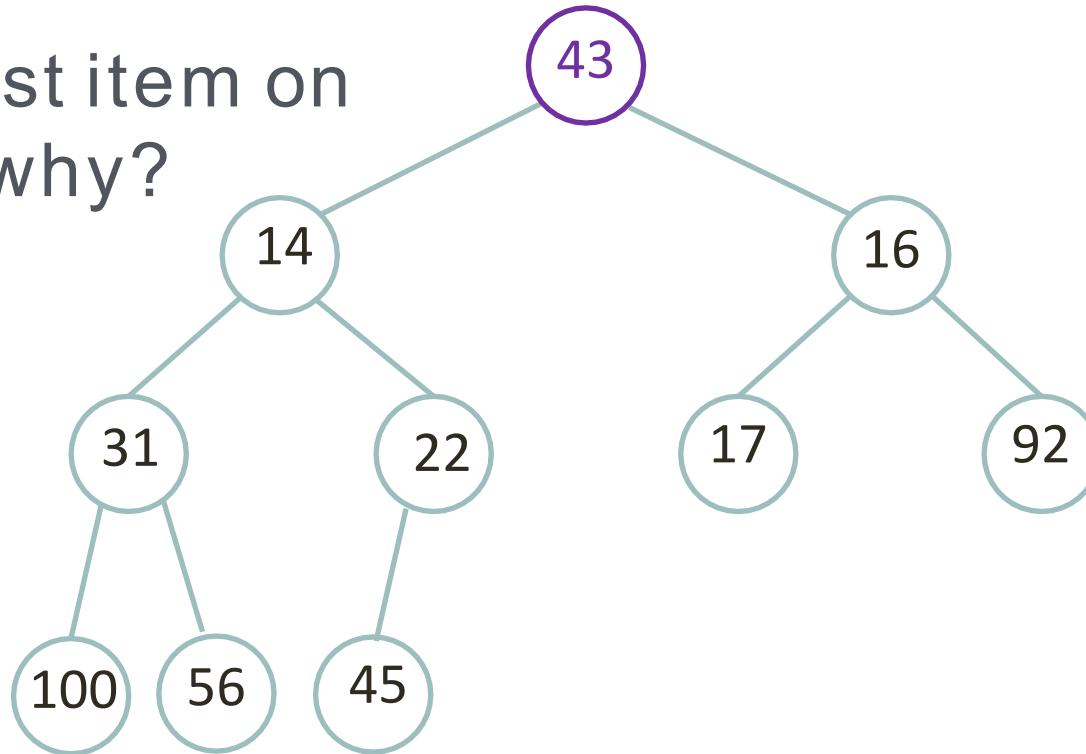


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down

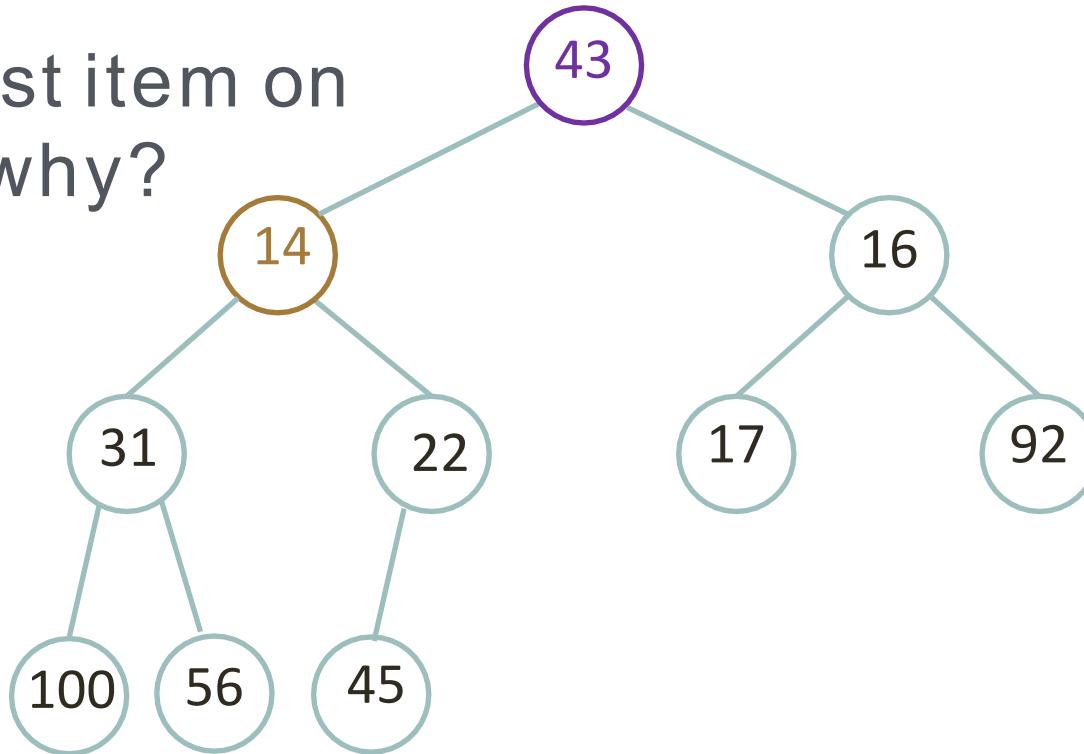


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down

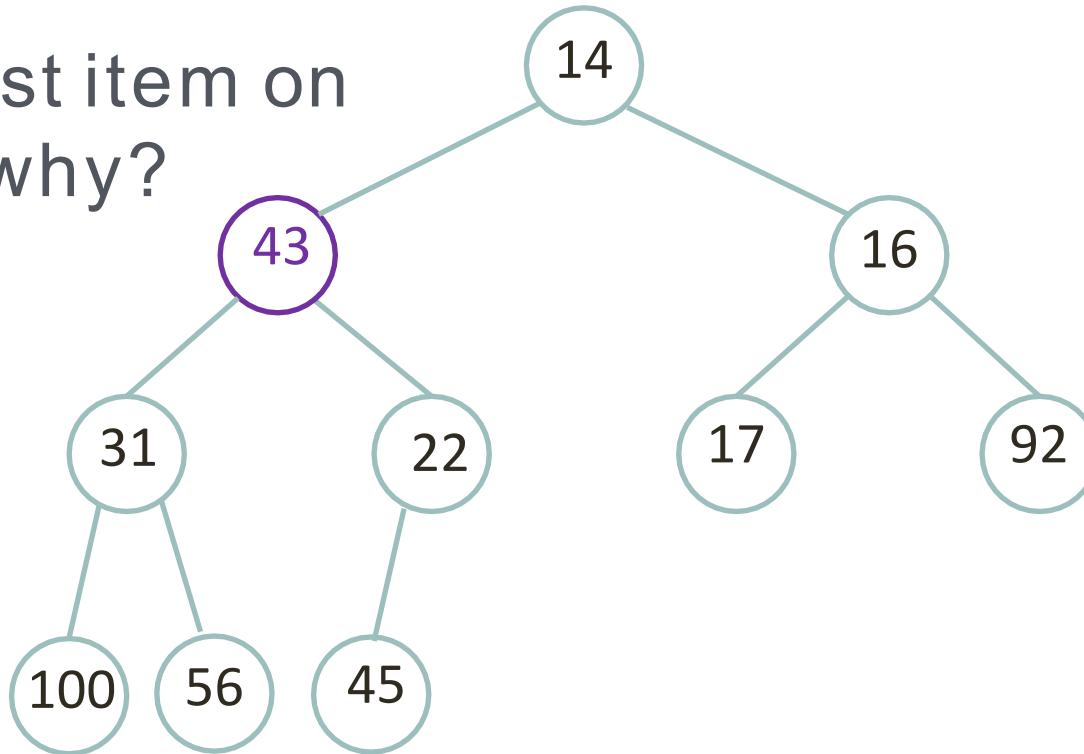


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down

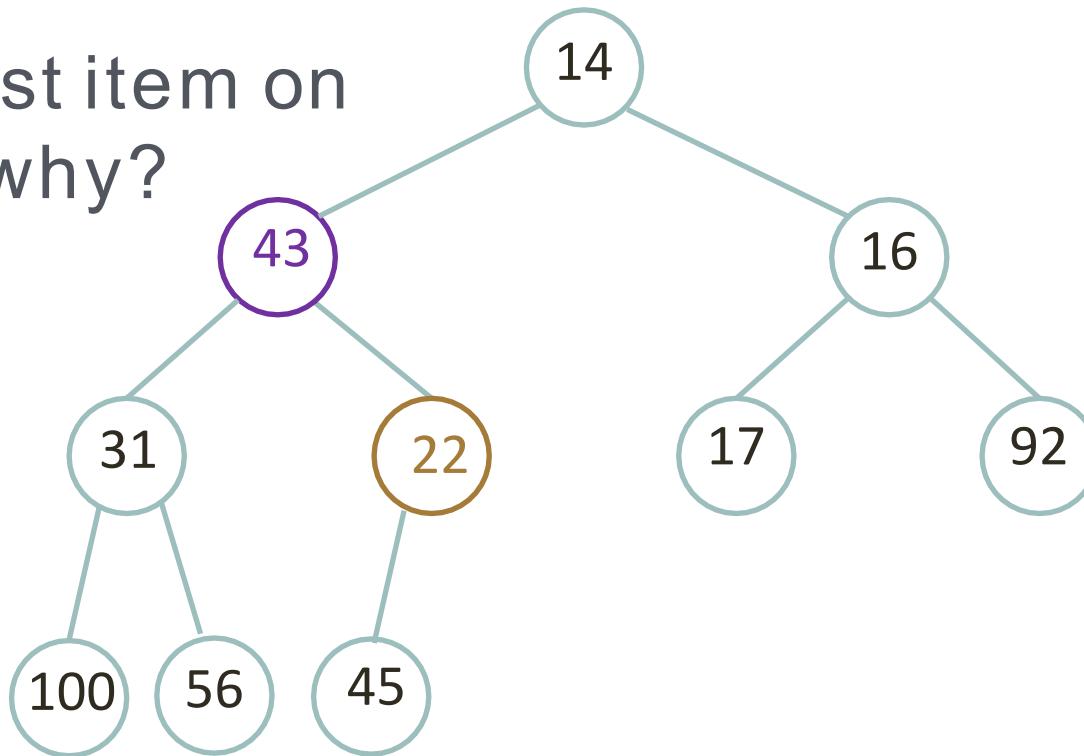


# Let's remove the smallest item

Take out 3

Fill with last item on last level. why?

Percolate down

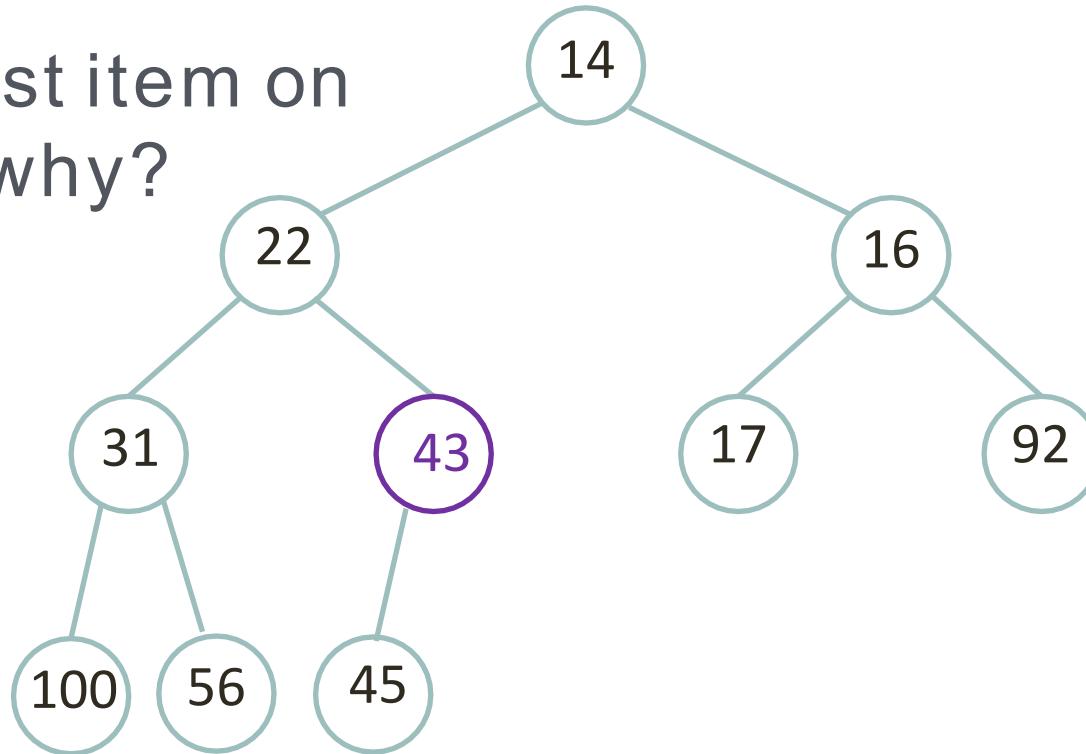


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down

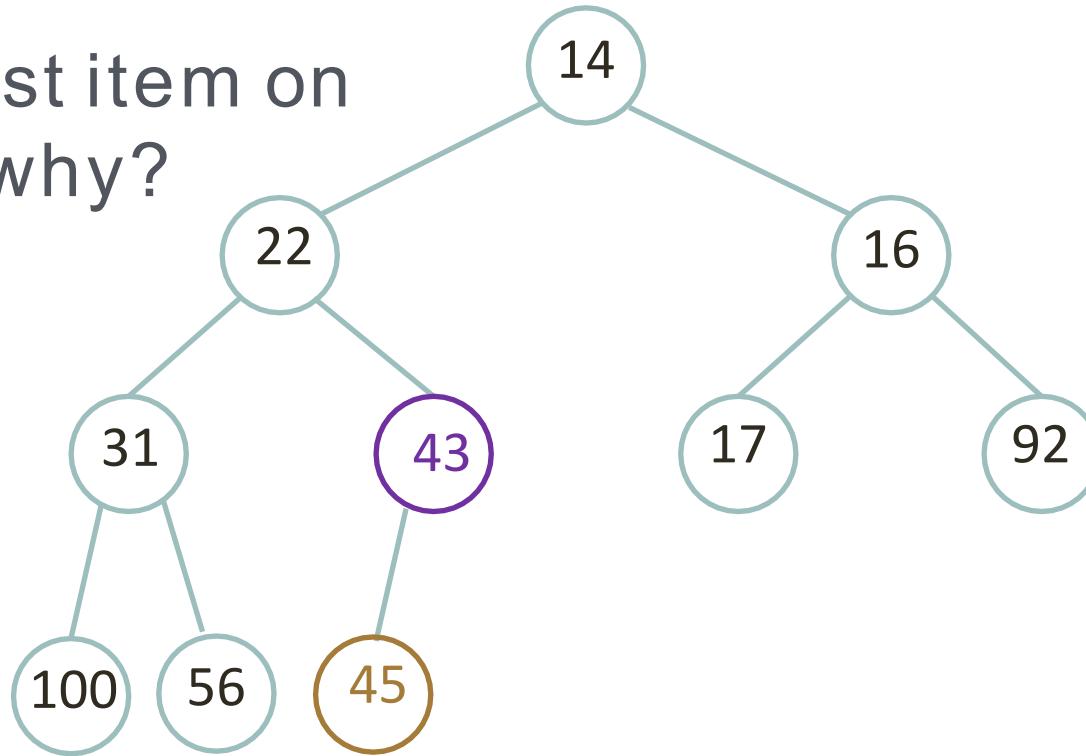


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down

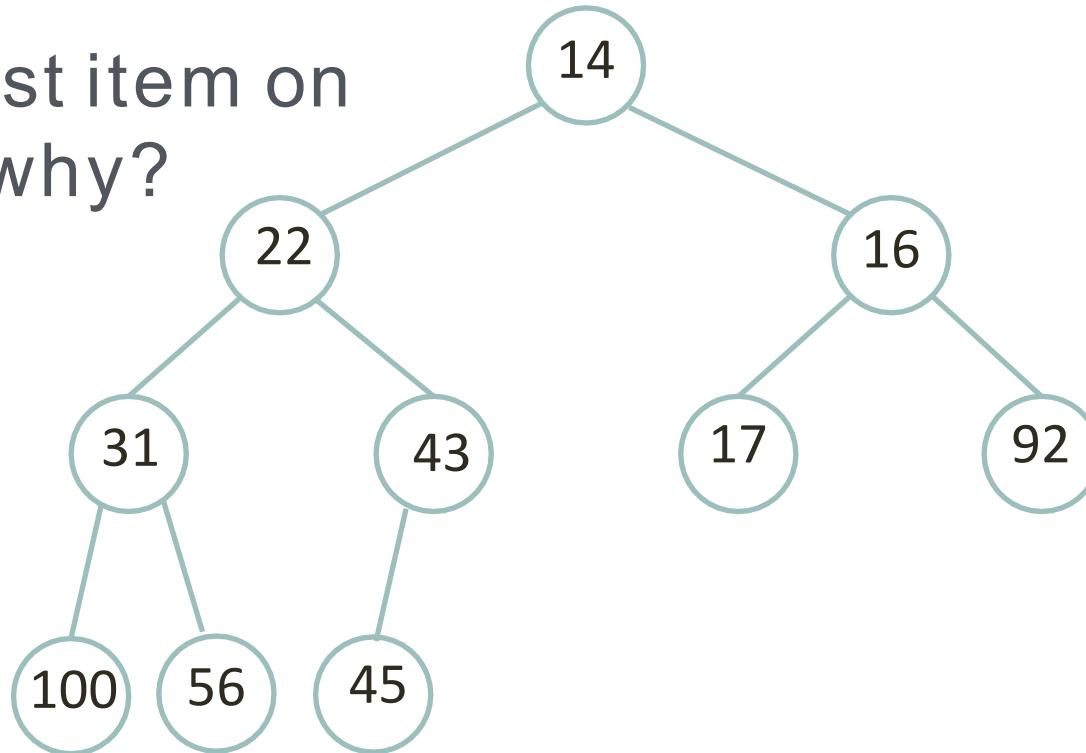


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down



# Cost of remove

- Worst case is  $O(\log N)$ 
  - Percolating down to the bottom level
- Average case is also  $O(\log N)$ 
  - Rarely terminates more than 1-2 levels from the bottom...

# Pseudo code...

Find and remove rightmost leaf node

Insert leaf's value into root node

Current node = root

While (current node's value is not smaller than both children) {

    choose smaller of two children.

    swap current node's value with smaller child value

    current node = smaller child node

}

# Recap

- Priority queues can be implemented any number of ways
- A binary heap's main use is for implementing priority queues
- Remember, the basic priority queue operations are:
  - add
  - findMin
  - deleteMin

- The average cases for a priority queue implemented with a binary heap:
  - add
    - **O(1)**: *percolate up (average of 2.6 compares)*
  - findMin
    - **O(1)**: *just return the root*
  - deleteMin
    - **O(logN)**: *percolate down (rarely terminates before near the bottom of the tree)*

# Heap implementation

- As a linked list
  - How to look for parent during bubbling?
  - How to access the rightmost leaf?

# Heap implementation

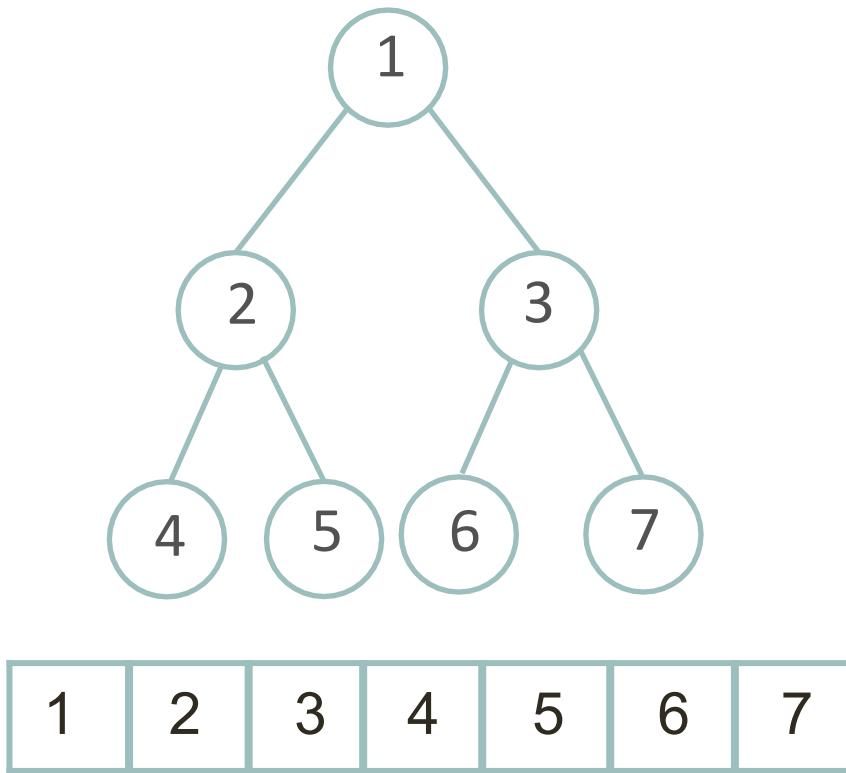
- As a linked list
  - How to look for parent during bubbling?
  - How to access the rightmost leaf?

Any complete tree can be stored as an array!

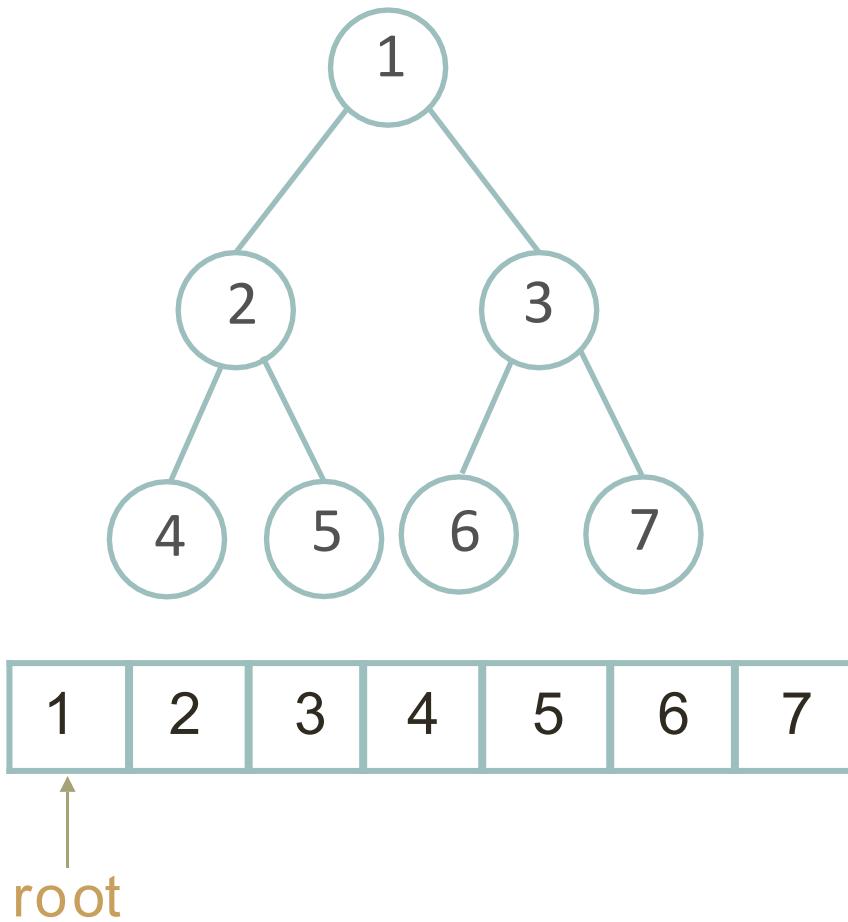
# complete trees as an array

- If we are guaranteed that tree is complete, we can implement it as an array instead of a linked structure
- The root goes at index 0, its left child at index 1, its right child at index 2
- For any node at index  $i$ , it's two children are at index  $(i*2) + 1$  and  $(i*2) + 2$

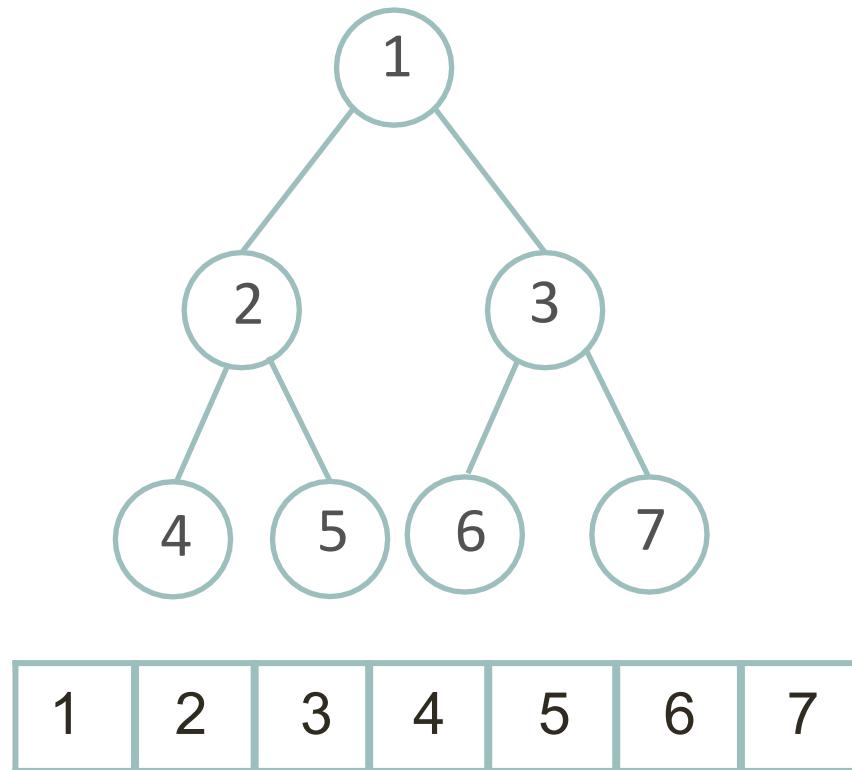
# Array heap implementation



# Array heap implementation

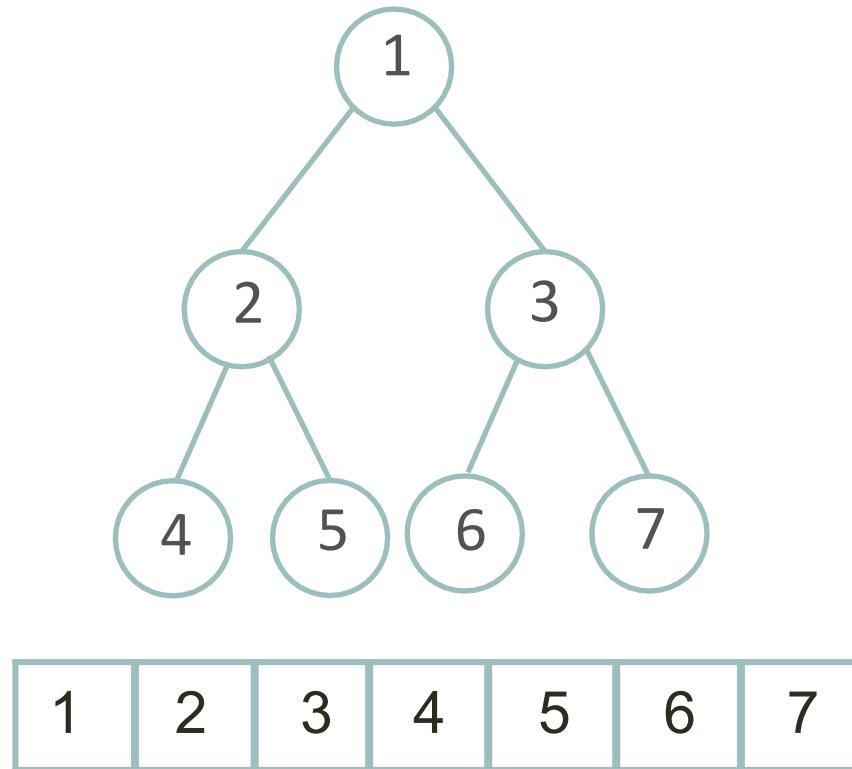


# Array heap implementation

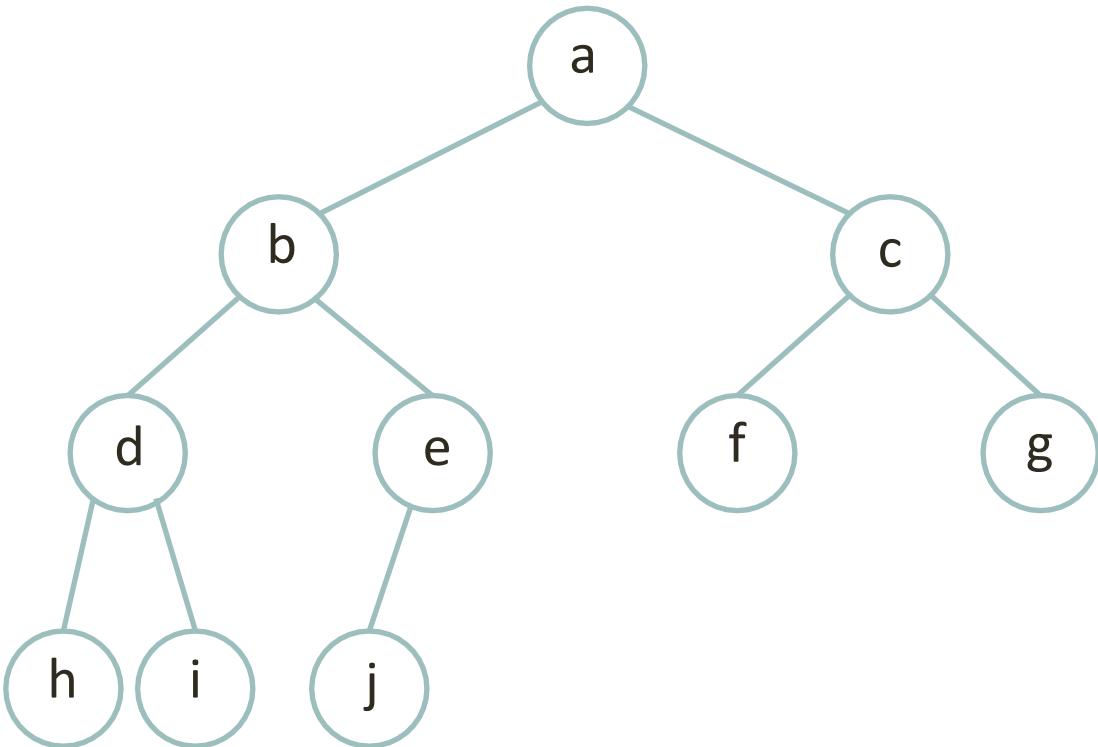


The left child of index  $i$  is stored at index  $2i+1$

# Array heap implementation

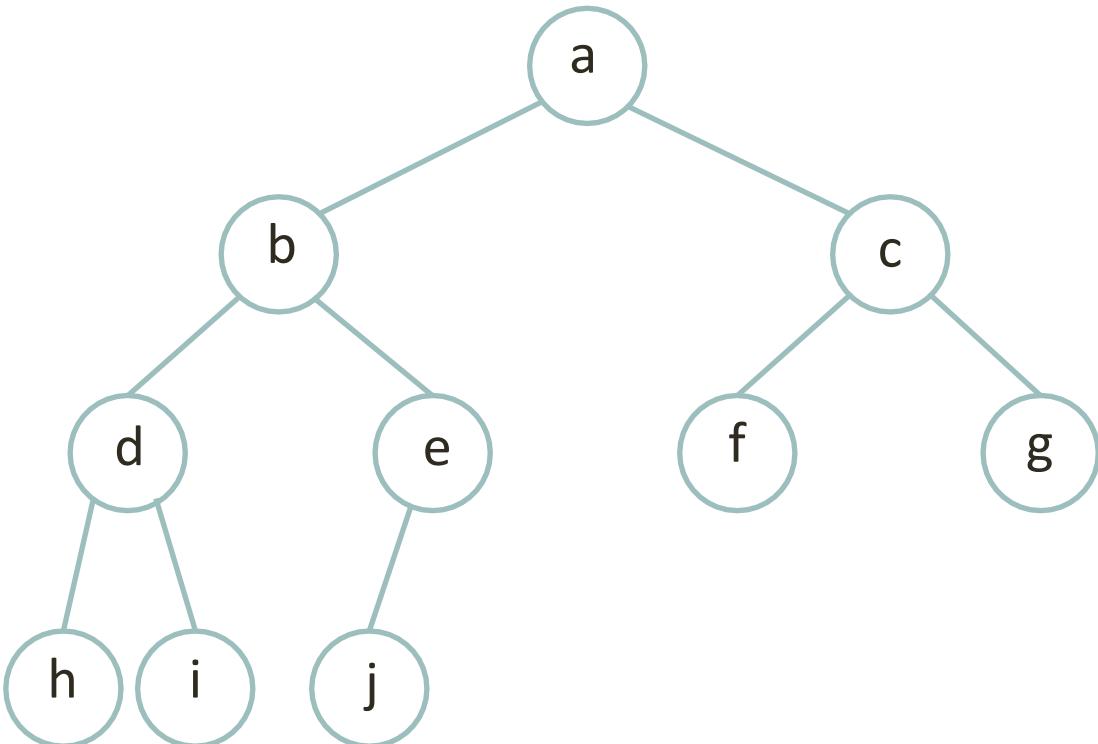


The right child of index  $i$  is stored at index  $2i+2$



| array | a | b | c | d | e | f | g | h | i | j |  |
|-------|---|---|---|---|---|---|---|---|---|---|--|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |

- How can we compute the index of *any* node's parent?



|       |   |   |   |   |   |   |   |   |   |   |  |
|-------|---|---|---|---|---|---|---|---|---|---|--|
| array | a | b | c | d | e | f | g | h | i | j |  |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |

- Luckily, integer division automatically truncates
- Any node's parent is at index  $(i-1) / 2$

# Complete trees as an array

- Keep track of a `currentSize` variable
  - Holds the total number of nodes in the tree
  - The very last leaf of the bottom level will be at index `currentSize - 1`
- When computing the index of a child node, if that index is  $\geq \text{currentSize}$ , then the child does not exist

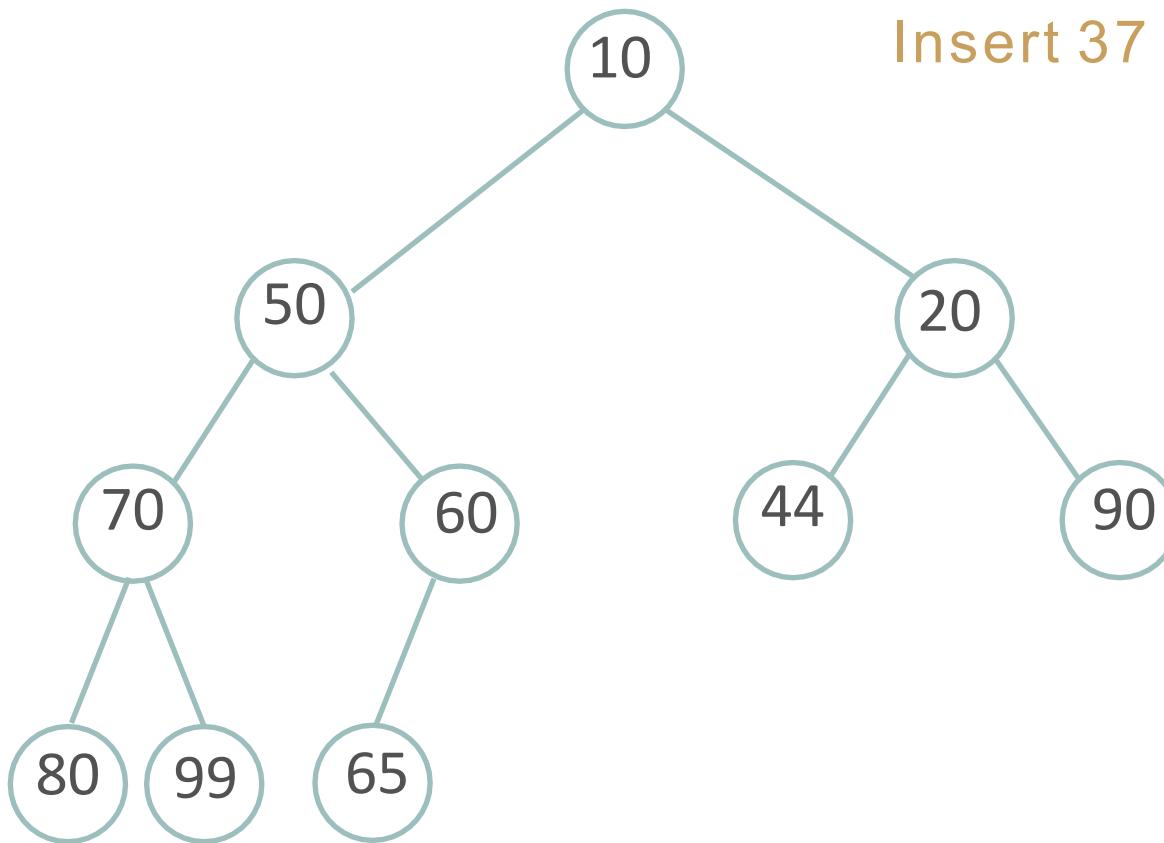
# traversal helper methods

```
int leftChildIndex(int i) {  
    return (i*2) + 1;  
}
```

```
int rightChildIndex(int i) {  
    return (i*2) + 2;  
}
```

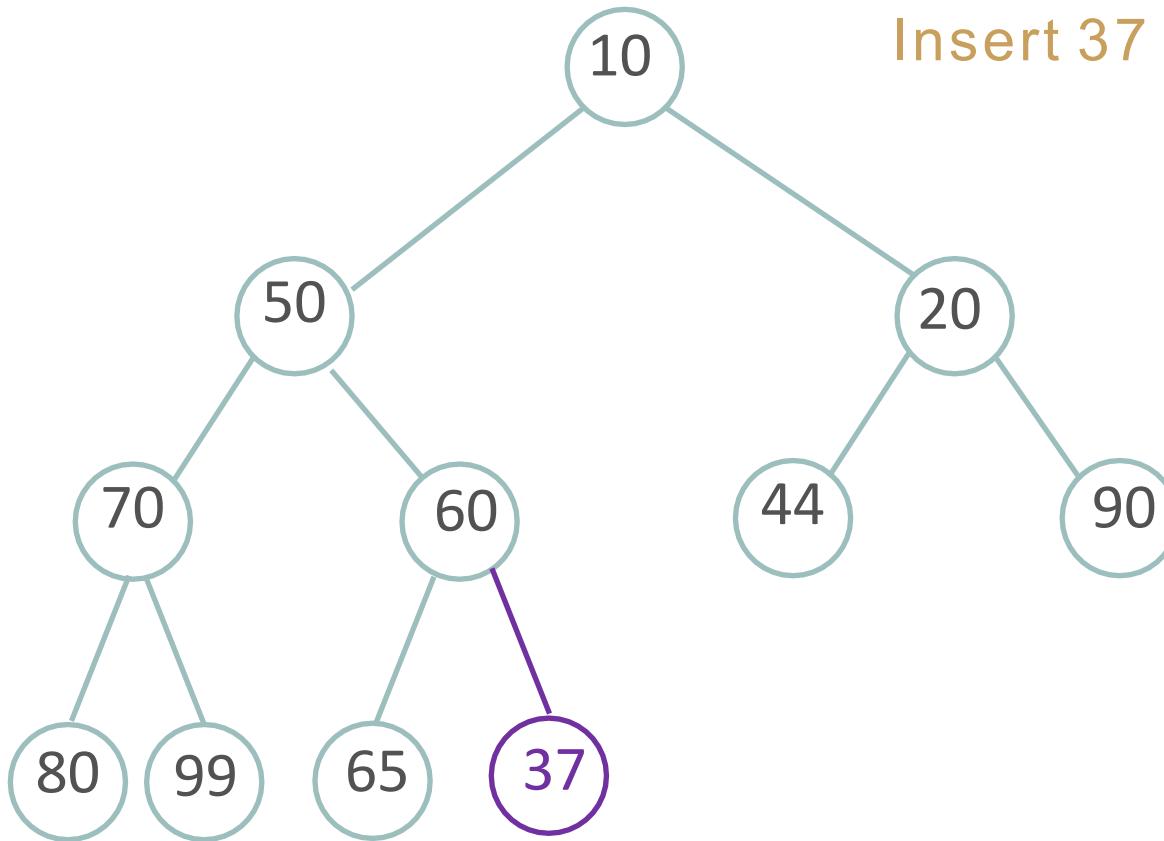
```
int parentIndex(int i) {  
    return (i-1) / 2;  
}
```

# Add



|    |    |    |    |    |    |    |    |    |    |  |  |
|----|----|----|----|----|----|----|----|----|----|--|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 |  |  |
|----|----|----|----|----|----|----|----|----|----|--|--|

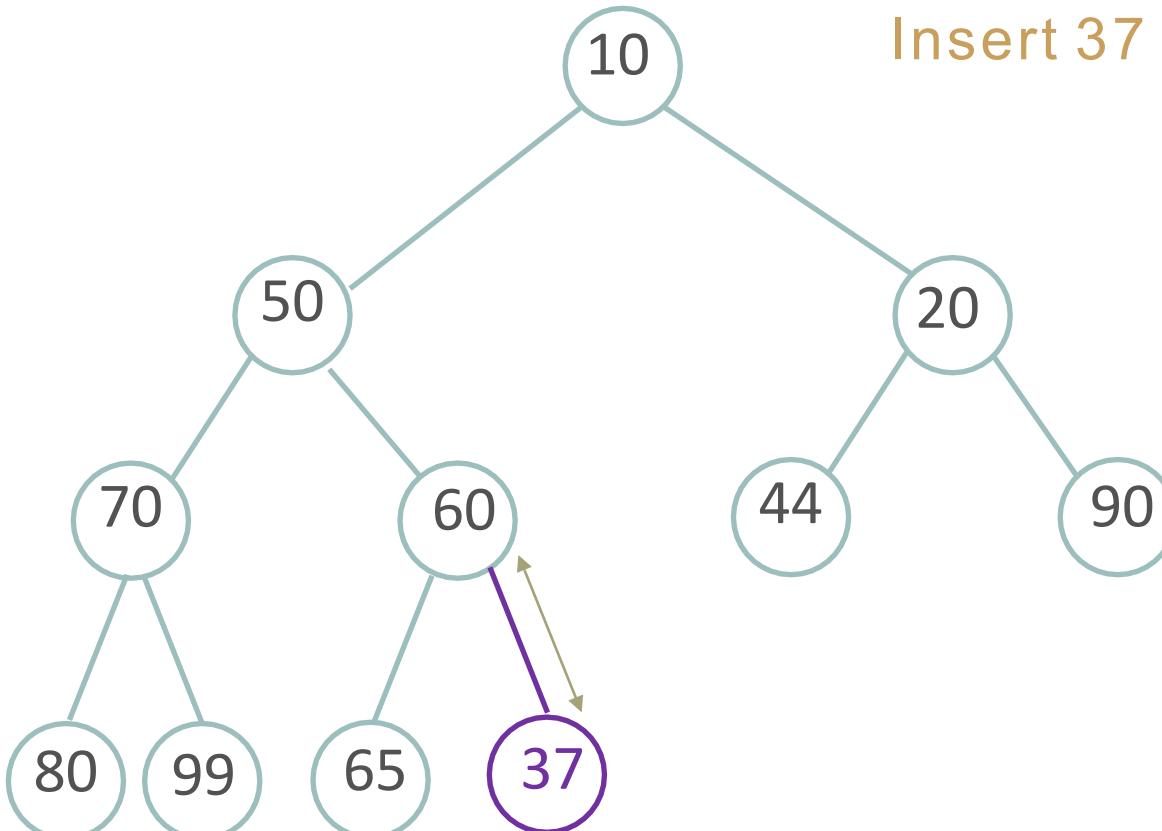
# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 | 37 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

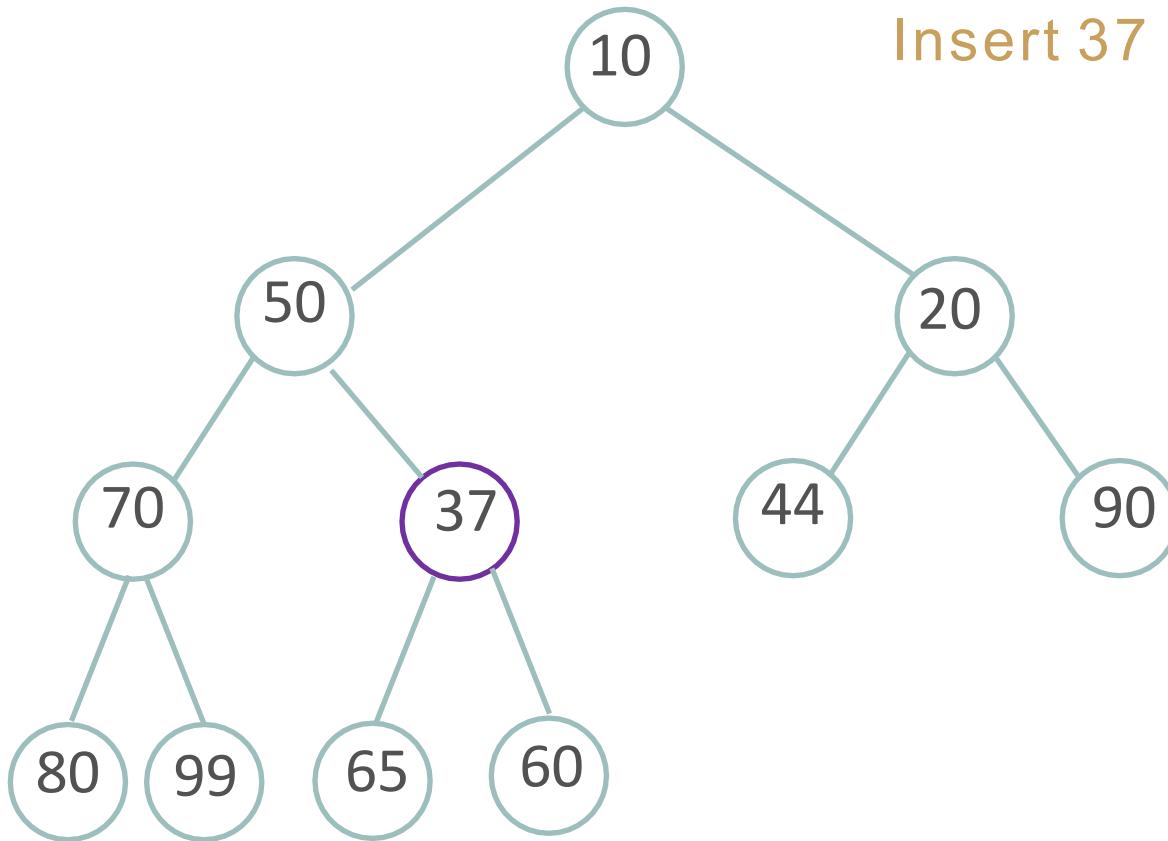
# Add

Insert 37



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 | 37 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

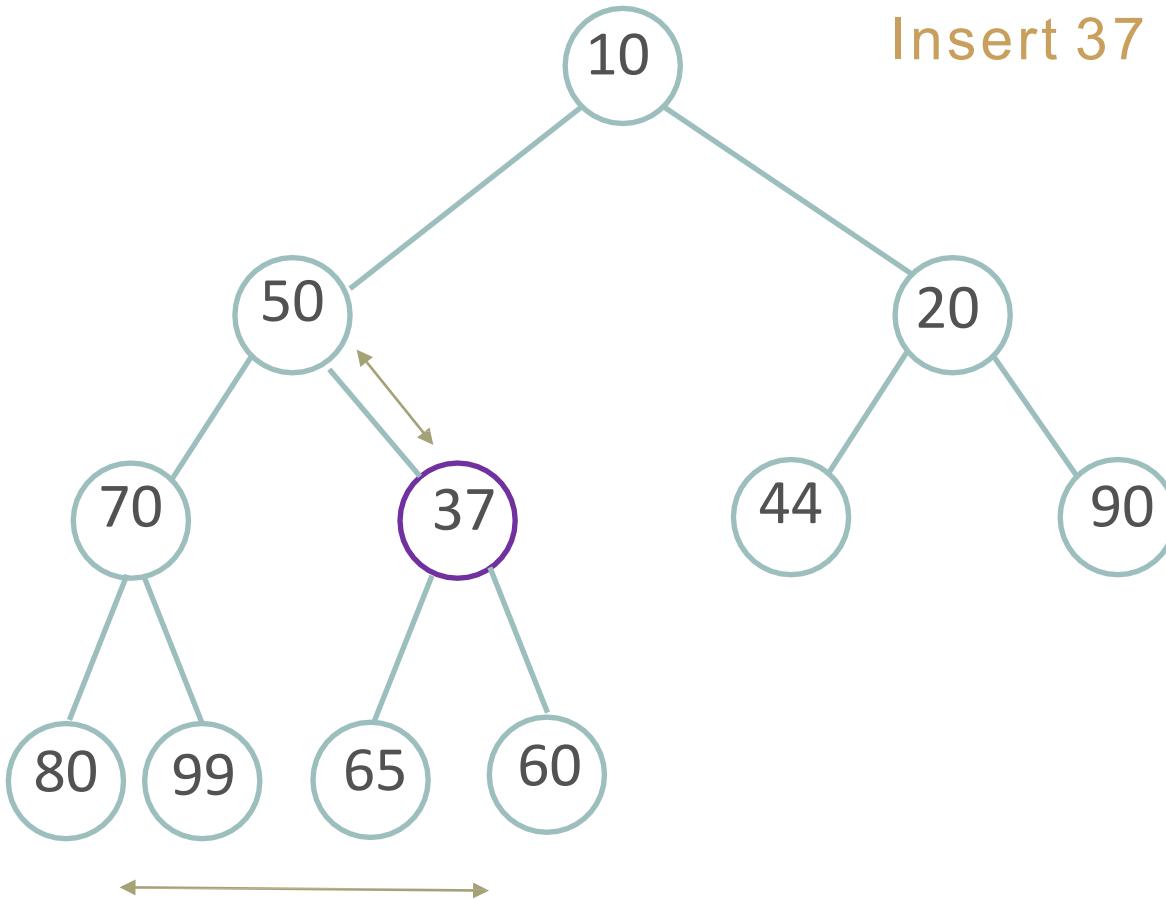
# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 37 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

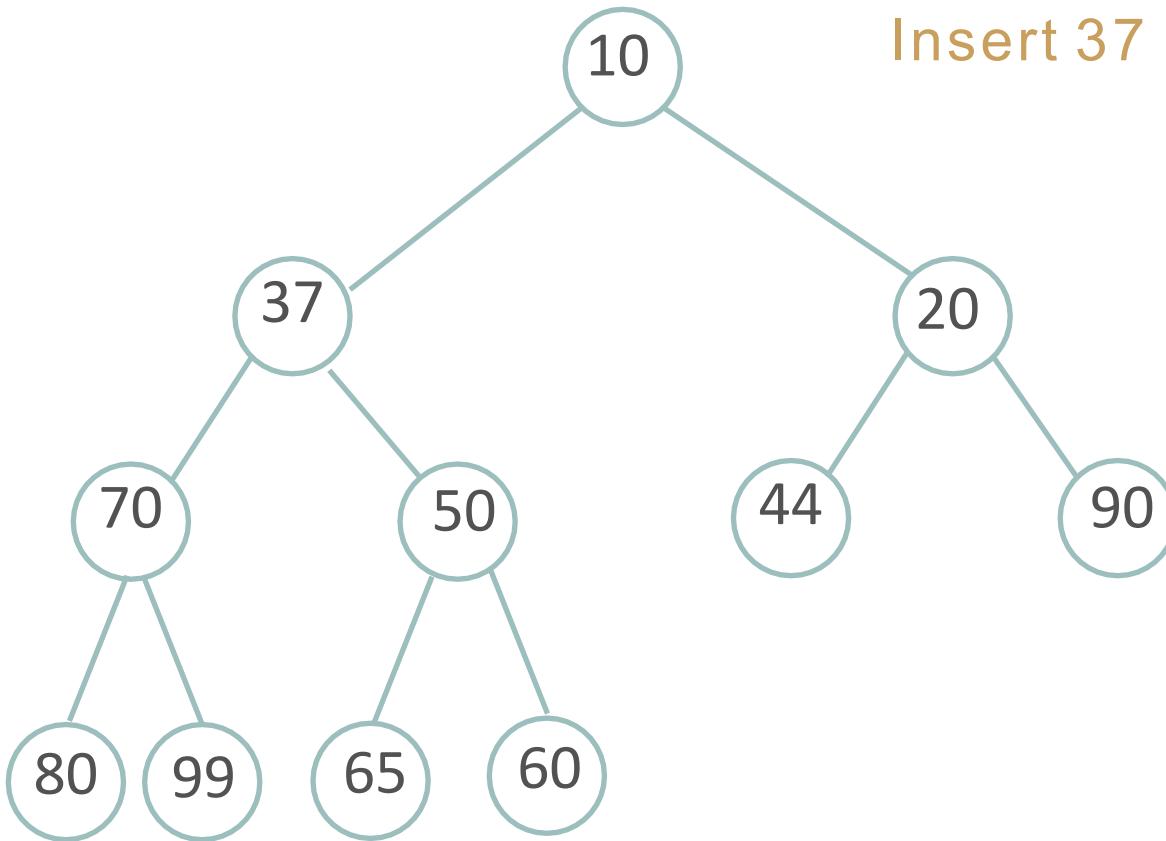
# Add

Insert 37



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 37 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 37 | 20 | 70 | 50 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

Remove is similar...

```
1 // Java implementation of Min Heap
2 public class MinHeap {
3     private int[] Heap;
4     private int size;
5     private int maxsize;
6
7     private static final int FRONT = 1;
8
9     public MinHeap(int maxsize)
10    {
11        this.maxsize = maxsize;
12        this.size = 0;
13        Heap = new int[this.maxsize + 1];
14        Heap[0] = Integer.MIN_VALUE;
15    }
16
17    // Function to return the position of
18    // the parent for the node currently
19    // at pos
20    private int parent(int pos)
21    {
22        return pos / 2;
23    }
24
25    // Function to return the position of the
26    // left child for the node currently at pos
27    private int leftChild(int pos)
28    {
29        return (2 * pos);
30    }
31
32    // Function to return the position of
33    // the right child for the node currently
34    // at pos
35    private int rightChild(int pos)
36    {
37        return (2 * pos) + 1;
38    }

```

```
39
40 // Function that returns true if the passed
41 // node is a leaf node
42 private boolean isLeaf(int pos)
43 {
44     if (pos >= (size / 2) && pos <= size) {
45         return true;
46     }
47     return false;
48 }
49
50 // Function to swap two nodes of the heap
51 private void swap(int fpos, int spos)
52 {
53     int tmp;
54     tmp = Heap[fpos];
55     Heap[fpos] = Heap[spos];
56     Heap[spos] = tmp;
57 }
58
59 // Function to heapify the node at pos
60 private void minHeapify(int pos)
61 {
62
63     // If the node is a non-leaf node and greater
64     // than any of its child
65     if (!isLeaf(pos)) {
66         if (Heap[pos] > Heap[leftChild(pos)]
67             || Heap[pos] > Heap[rightChild(pos)]) {
68
69             // Swap with the left child and heapify
70             // the left child
71             if (Heap[leftChild(pos)] < Heap[rightChild(pos)]) {
72                 swap(pos, leftChild(pos));
73                 minHeapify(leftChild(pos));
74             }
75
76             // Swap with the right child and heapify
77             // the right child
78             else {
79                 swap(pos, rightChild(pos));
80                 minHeapify(rightChild(pos));
81             }
82         }
83     }
84 }
```

```
85
86 // Function to insert a node into the heap
87 public void insert(int element)
88 {
89     if (size >= maxsize) {
90         return;
91     }
92     Heap[++size] = element;
93     int current = size;
94
95     while (Heap[current] < Heap[parent(current)]) {
96         swap(current, parent(current));
97         current = parent(current);
98     }
99 }
100
101 // Function to print the contents of the heap
102 public void print()
103 {
104     for (int i = 1; i <= size / 2; i++) {
105         System.out.print(" PARENT : " + Heap[i]
106                         + " LEFT CHILD : " + Heap[2 * i]
107                         + " RIGHT CHILD :" + Heap[2 * i + 1]);
108         System.out.println();
109     }
110 }
111
112 // Function to build the min heap using
113 // the minHeapify
114 public void minHeap()
115 {
116     for (int pos = (size / 2); pos >= 1; pos--) {
117         minHeapify(pos);
118     }
119 }
```

```
120
121     // Function to remove and return the minimum
122     // element from the heap
123     public int remove()
124     {
125         int popped = Heap[FRONT];
126         Heap[FRONT] = Heap[size--];
127         minHeapify(FRONT);
128         return popped;
129     }
130
131     // Driver code
132     public static void main(String[] arg)
133     {
134         System.out.println("The Min Heap is ");
135         MinHeap minHeap = new MinHeap(15);
136         minHeap.insert(5);
137         minHeap.insert(3);
138         minHeap.insert(17);
139         minHeap.insert(10);
140         minHeap.insert(84);
141         minHeap.insert(19);
142         minHeap.insert(6);
143         minHeap.insert(22);
144         minHeap.insert(9);
145         minHeap.minHeap();
146
147         minHeap.print();
148         System.out.println("The Min val is " + minHeap.remove());
149     }
150 }
```

```
1 import java.util.Arrays;
2 import java.util.Vector;
3
4 // class for implementing Priority Queue
5 class PriorityQueue
6 {
7     // vector to store heap elements
8     private Vector<Integer> A;
9
10    // constructor: use default initial capacity of vector
11    public PriorityQueue()
12    {
13        A = new Vector();
14    }
15
16    // constructor: set custom initial capacity for vector
17    public PriorityQueue(int capacity)
18    {
19        A = new Vector(capacity);
20    }
21
22    // return parent of A.get(i)
23    private int parent(int i)
24    {
25        // if i is already a root node
26        if (i == 0)
27            return 0;
28
29        return (i - 1) / 2;
30    }
}
```

```
31 // return left child of A.get(i)
32 private int LEFT(int i)
33 {
34     return (2 * i + 1);
35 }
36
37 // return right child of A.get(i)
38 private int RIGHT(int i)
39 {
40     return (2 * i + 2);
41 }
42
43
44 // swap values at two indexes
45 void swap(int x, int y)
46 {
47     // swap with child having greater value
48     Integer temp = A.get(x);
49     A.setElementAt(A.get(y), x);
50     A.setElementAt(temp, y);
51 }
52
53 // Recursive Heapify-down procedure. Here the node at index i
54 // and its two direct children violates the heap property
55 private void heapify_down(int i)
56 {
57     // get left and right child of node at index i
58     int left = LEFT(i);
59     int right = RIGHT(i);
60
61     int smallest = i;
62
63     // compare A.get(i) with its left and right child
64     // and find smallest value
65     if (left < size() && A.get(left) < A.get(i)) {
66         smallest = left;
67     }
68
69     if (right < size() && A.get(right) < A.get(smallest)) {
70         smallest = right;
71     }
72
73     if (smallest != i)
74     {
75         // swap with child having lesser value
76         swap(i, smallest);
77
78         // call heapify-down on the child
79         heapify_down(smallest);
80     }
81 }
```

```
82
83     // Recursive Heapify-up procedure
84     private void heapify_up(int i)
85     {
86         // check if node at index i and its parent violates
87         // the heap property
88         if (i > 0 && A.get(parent(i)) > A.get(i))
89         {
90             // swap the two if heap property is violated
91             swap(i, parent(i));
92
93             // call Heapify-up on the parent
94             heapify_up(parent(i));
95         }
96     }
97
98     // return size of the heap
99     public int size()
100    {
101        return A.size();
102    }
103
104    // check if heap is empty or not
105    public Boolean isEmpty()
106    {
107        return A.isEmpty();
108    }
109
110    // insert specified key into the heap
111    public void add(Integer key)
112    {
113        // insert the new element to the end of the vector
114        A.addElement(key);
115
116        // get element index and call heapify-up procedure
117        int index = size() - 1;
118        heapify_up(index);
119    }
```

```
120
121 // function to remove and return element with highest priority
122 // (present at root). It returns null if queue is empty
123 public Integer poll()
124 {
125     try {
126         // if heap is empty, throw an exception
127         if (size() == 0) {
128             throw new Exception("Index is out of range (Heap underflow)");
129         }
130
131         // element with highest priority
132         int root = A.firstElement();    // or A.get(0);
133
134         // replace the root of the heap with the last element of the vector
135         A.setElementAt(A.lastElement(), 0);
136         A.remove(size()-1);
137
138         // call heapify-down on root node
139         heapify_down(0);
140
141         // return root element
142         return root;
143     }
144     // catch and print the exception
145     catch (Exception ex) {
146         System.out.println(ex);
147         return null;
148     }
149 }
150
151 // function to return, but does not remove, element with highest priority
152 // (present at root). It returns null if queue is empty
153 public Integer peek()
154 {
155     try {
156         // if heap has no elements, throw an exception
157         if (size() == 0) {
158             throw new Exception("Index out of range (Heap underflow)");
159         }
160
161         // else return the top (first) element
162         return A.firstElement();    // or A.get(0);
163     }
164     // catch the exception and print it, and return null
165     catch (Exception ex) {
166         System.out.println(ex);
167         return null;
168     }
169 }
```

```
170
171 // function to remove all elements from priority queue
172 public void clear()
173 {
174     System.out.print("Emptying queue: ");
175     while (!A.isEmpty()) {
176         System.out.print(poll() + " ");
177     }
178     System.out.println();
179 }
180
181 // returns true if queue contains the specified element
182 public Boolean contains(Integer i)
183 {
184     return A.contains(i);
185 }
186
187 // returns an array containing all elements in the queue
188 public Integer[] toArray()
189 {
190     return A.toArray(new Integer[size()]);
191 }
192 }
```

```
194 class Main
195 {
196     // Program for Max Heap Implementation in Java
197     public static void main (String[] args)
198     {
199         // create a Priority Queue of initial capacity 10
200         // Priority of an element is decided by element's value
201         PriorityQueue pq = new PriorityQueue(10);
202
203         // insert three integers
204         pq.add(3);
205         pq.add(2);
206         pq.add(15);
207
208         // print Priority Queue size
209         System.out.println("Priority Queue Size is " + pq.size());
210
211         // search 2 in Priority Queue
212         Integer searchKey = 2;
213
214         if (pq.contains(searchKey)) {
215             System.out.println("Priority Queue contains " + searchKey + "\n");
216         }
217
218         // empty queue
219         pq.clear();
220
221         if (pq.isEmpty()) {
222             System.out.println("Queue is Empty");
223         }
224
225         System.out.println("\nCalling remove operation on an empty heap");
226         System.out.println("Element with highest priority is " + pq.poll() + '\n');
227
228         System.out.println("Calling peek operation on an empty heap");
229         System.out.println("Element with highest priority is " + pq.peek() + '\n');
230
231         // again insert three integers
232         pq.add(5);
233         pq.add(4);
234         pq.add(45);
235
236         // construct array containing all elements present in the queue
237         Integer[] I = pq.toArray();
238         System.out.println("Printing array: " + Arrays.toString(I));
239
240         System.out.println("\nElement with highest priority is " + pq.poll());
241         System.out.println("Element with highest priority is " + pq.peek());
242     }
243 }
```

Next time...

- Heap sort
- Quiz on Thursday
  - Trees and heaps
- Start your assignment early!

Heaps

CSC220 | Computer Programming 2

Last Time...

# Recap

- Heap versus priority queue
  - Priority queue: a description of a set of operations
  - Heap: one possible efficient way of implementing those operations

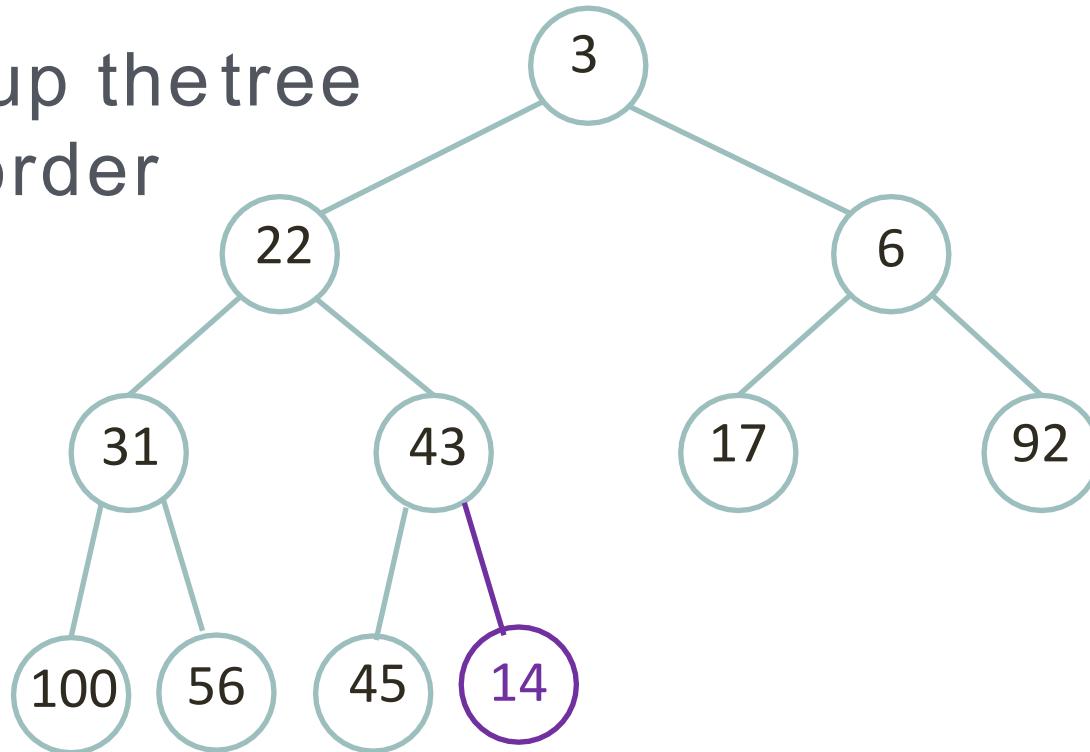
- A **binary heap** is a binary tree with two special properties
  - *Structure*: it is a complete tree
  - *Order*: the data in any node is less than or equal to the data of its children
- This is also called a **min-heap**
  - A **max-heap** would have the opposite property

# Add

# Adding 14

Put it at the end of the tree

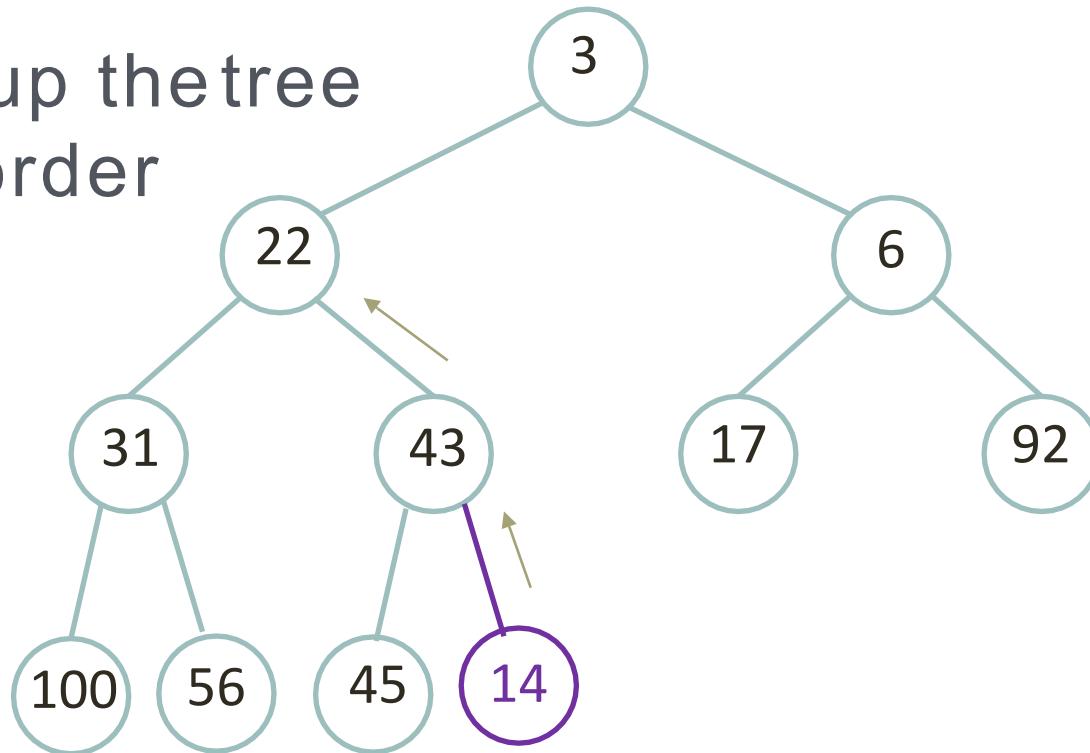
Percolate up the tree  
to fix the order



# Adding 14

Put it at the end of the tree

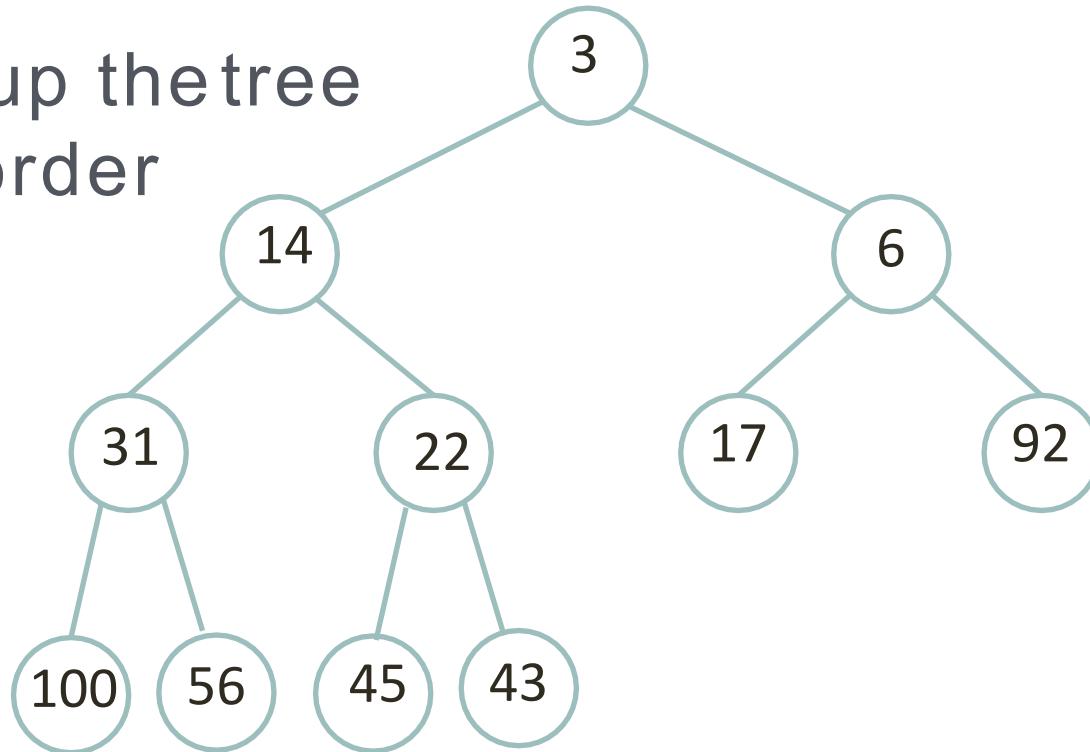
Percolate up the tree  
to fix the order



## Adding 14

Put it at the end of the tree

Percolate up the tree  
to fix the order

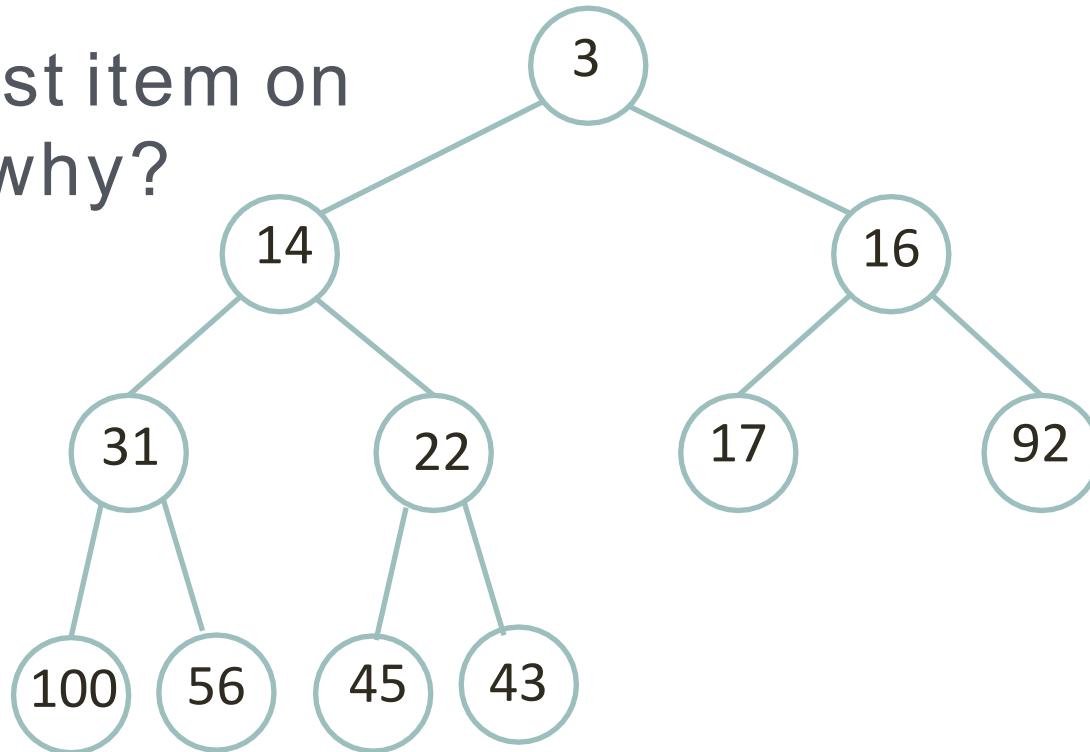


Remove

# Let's remove the smallest item

Take out 3

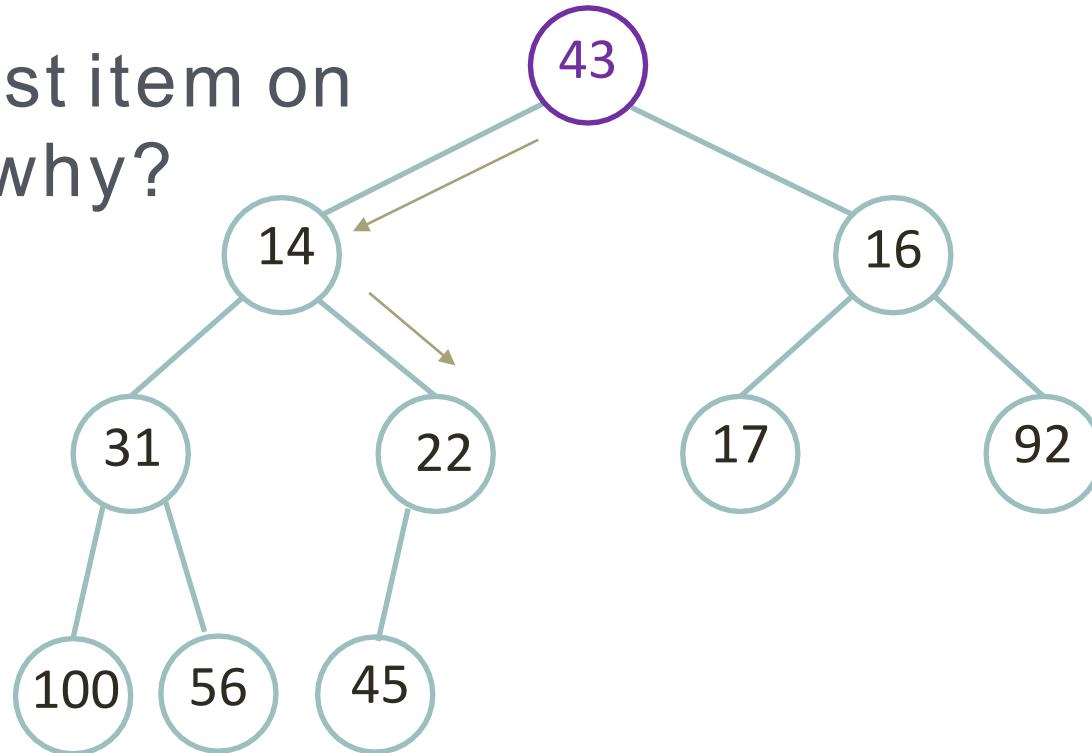
Fill with last item on  
last level. why?



# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

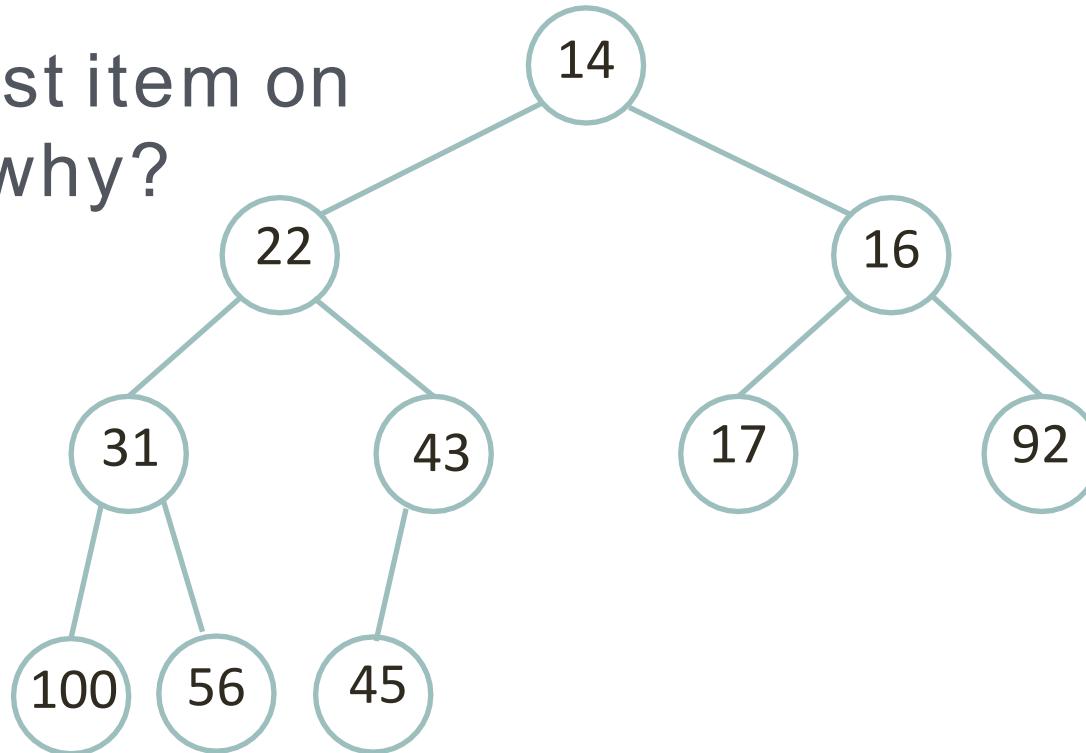


# Let's remove the smallest item

Take out 3

Fill with last item on  
last level. why?

Percolate  
down



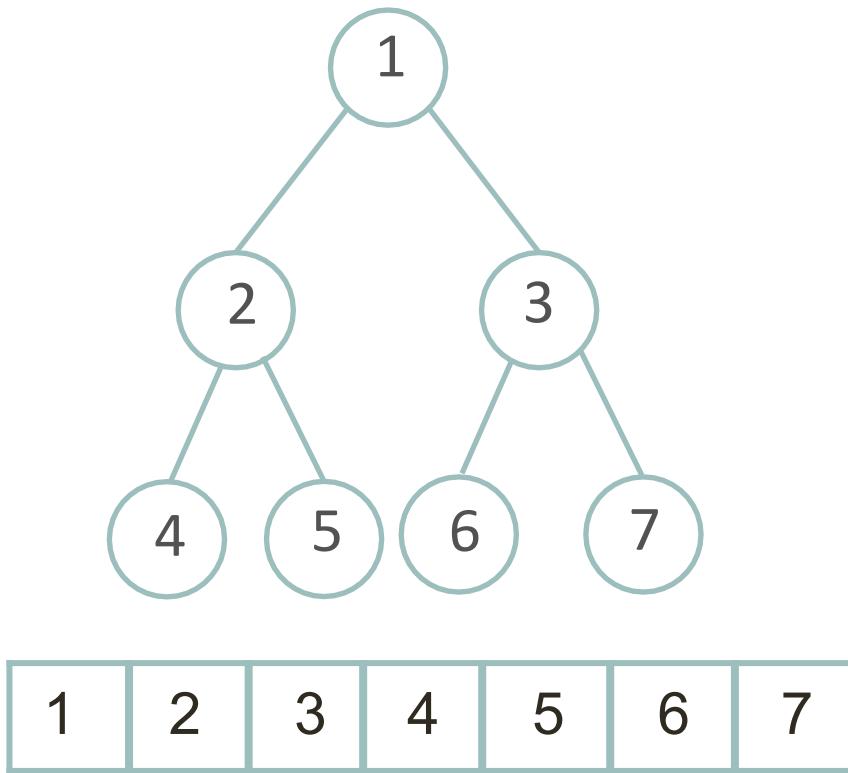
- The average cases for a priority queue implemented with a binary heap:
  - add
    - **O(1)**: *percolate up (average of 2.6 compares)*
  - findMin
    - **O(1)**: *just return the root*
  - deleteMin
    - **O(logN)**: *percolate down (rarely terminates before near the bottom of the tree)*

# Heap implementation

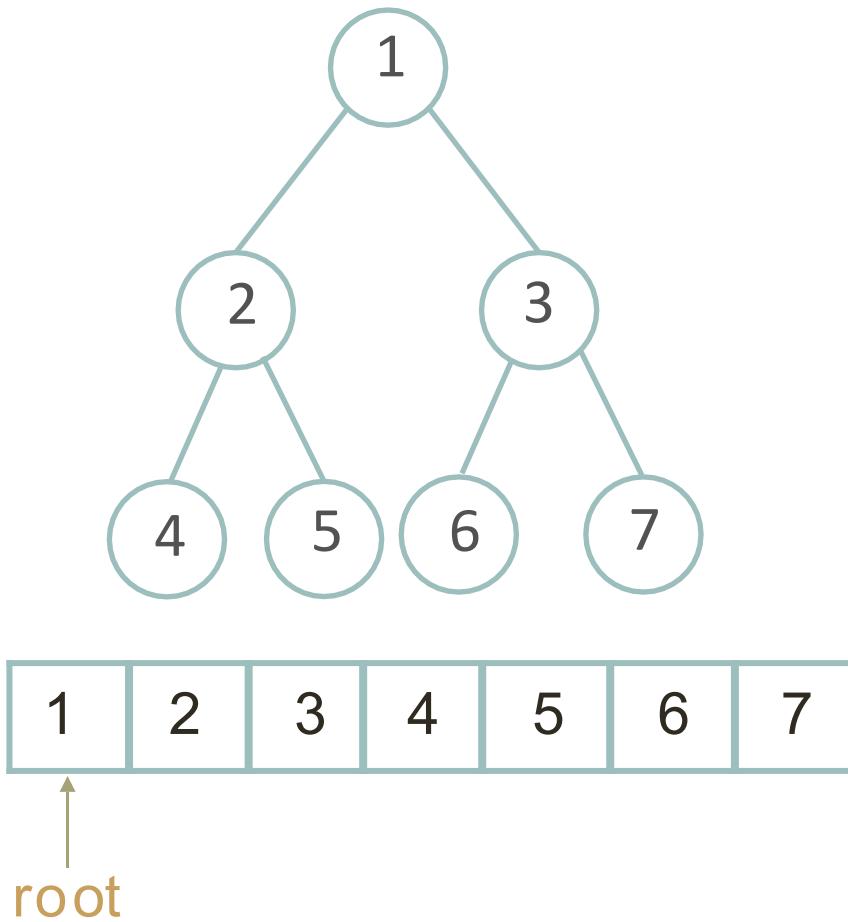
- As a linked list
  - How to look for parent during bubbling?
  - How to access the rightmost leaf?

Any complete tree can be stored as an array!

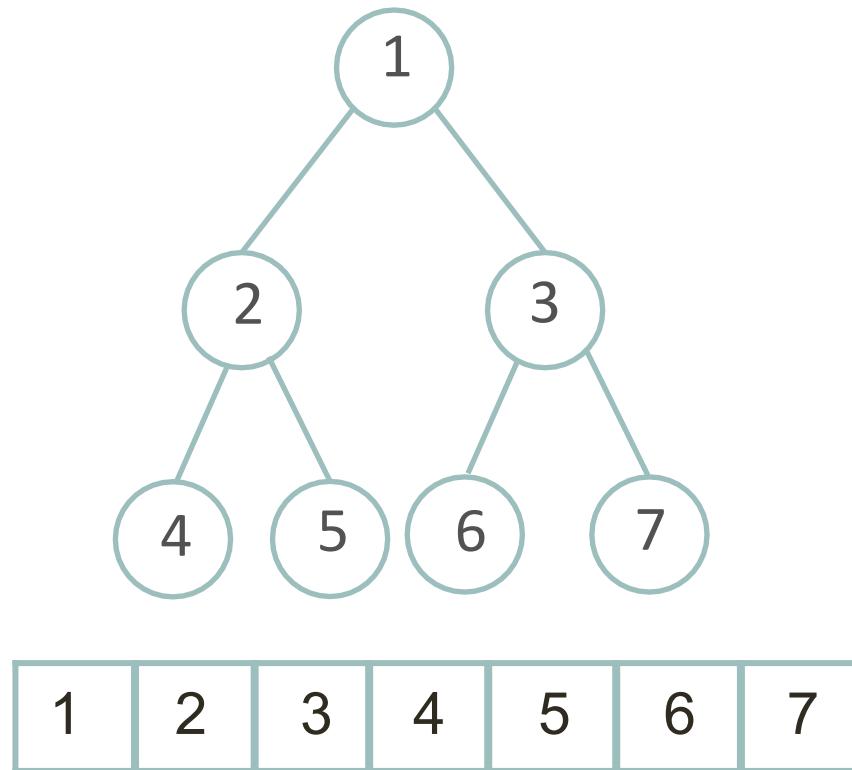
# Array heap implementation



# Array heap implementation

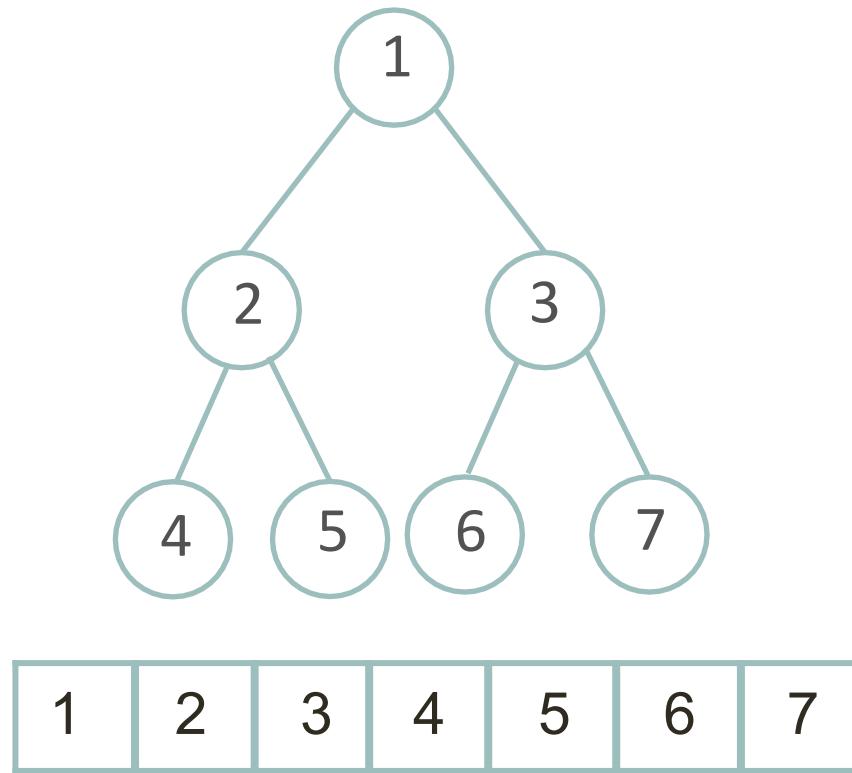


# Array heap implementation

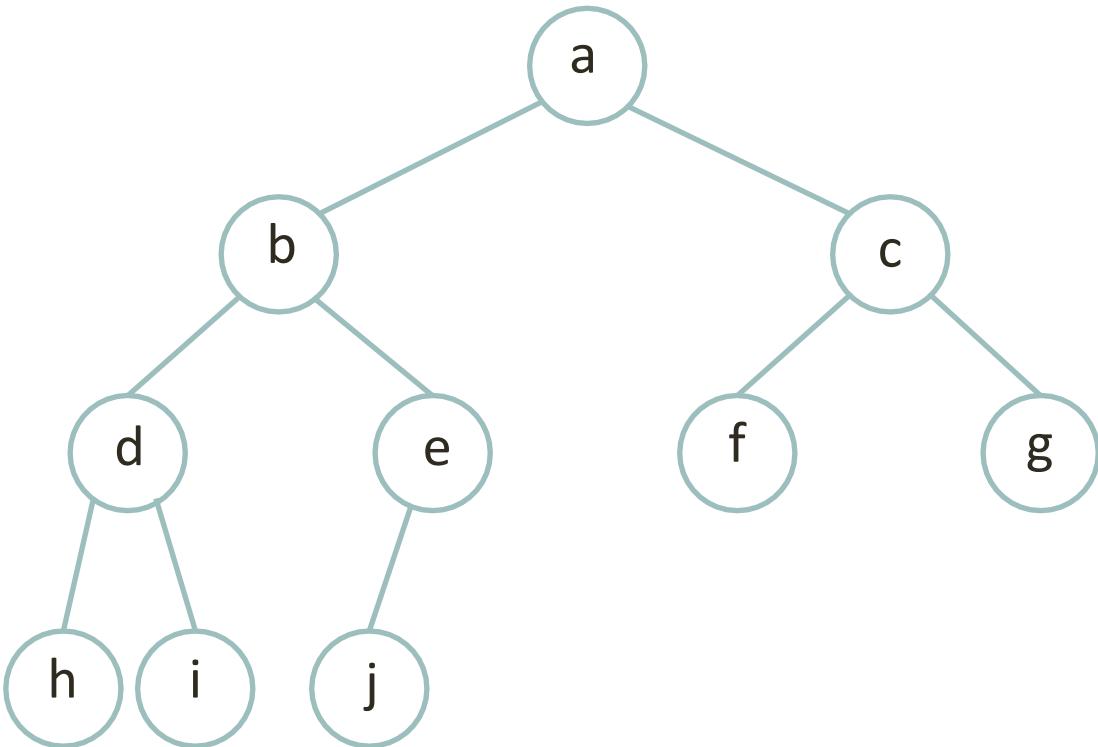


The left child of index  $i$  is stored at index  $2i+1$

# Array heap implementation

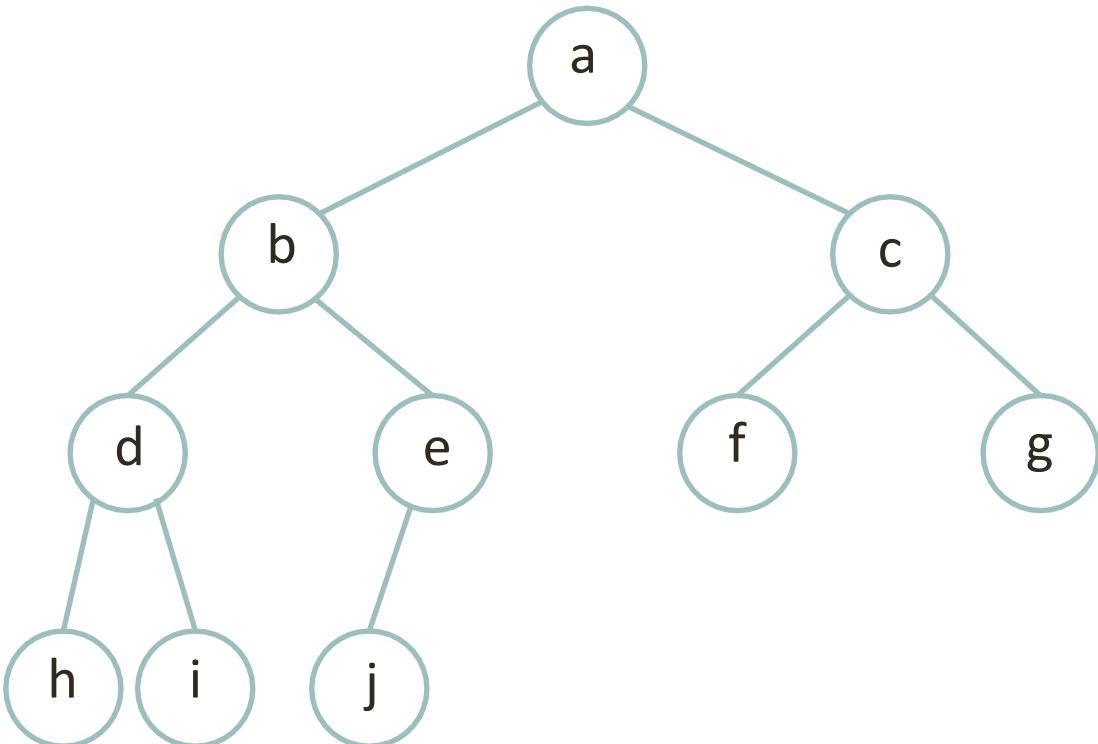


The right child of index  $i$  is stored at index  $2i+2$



| array | a | b | c | d | e | f | g | h | i | j |  |
|-------|---|---|---|---|---|---|---|---|---|---|--|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |

- How can we compute the index of *any* node's parent?



|       |   |   |   |   |   |   |   |   |   |   |  |
|-------|---|---|---|---|---|---|---|---|---|---|--|
| array | a | b | c | d | e | f | g | h | i | j |  |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |

- Luckily, integer division automatically truncates
- Any node's parent is at index  $(i-1) / 2$

# Complete trees as an array

- Keep track of a `currentSize` variable
  - Holds the total number of nodes in the tree
  - The very last leaf of the bottom level will be at index `currentSize - 1`
- When computing the index of a child node, if that index is  $\geq \text{currentSize}$ , then the child does not exist

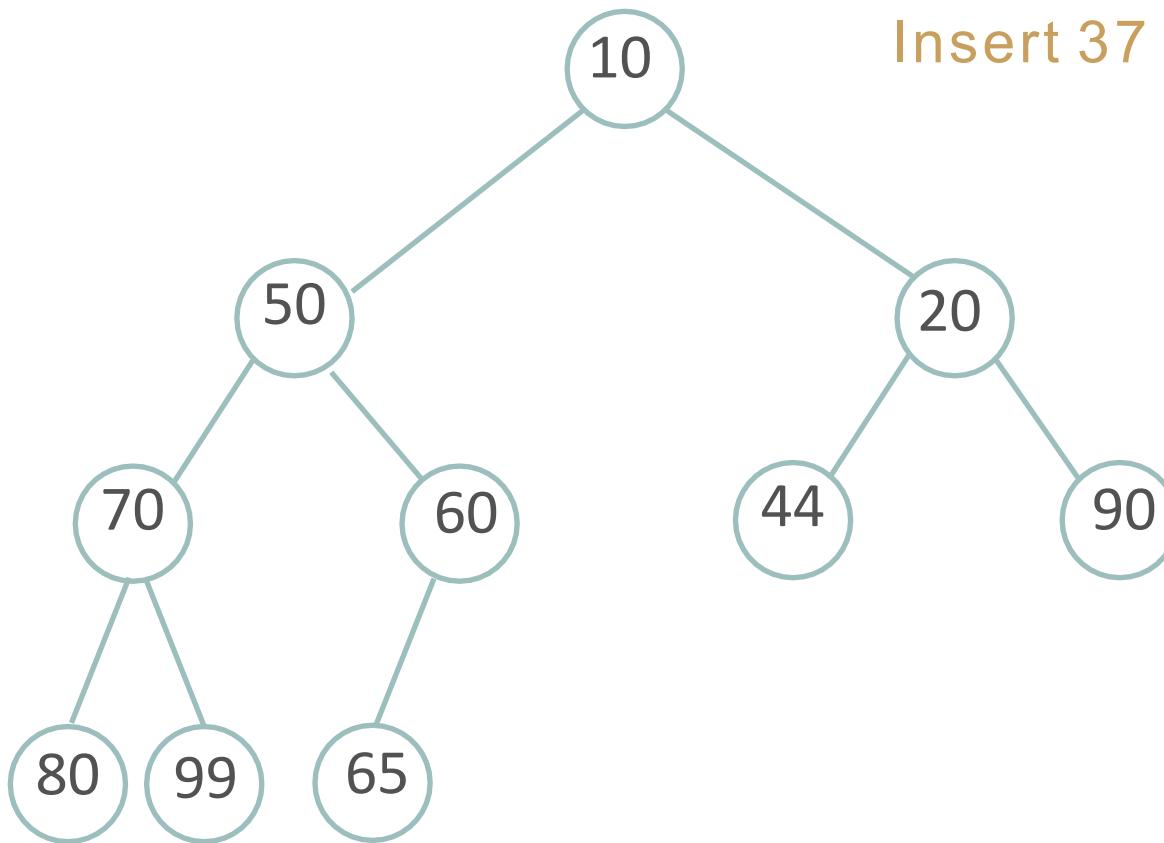
# traversal helper methods

```
int leftChildIndex(int i) {  
    return (i*2) + 1;  
}
```

```
int rightChildIndex(int i) {  
    return (i*2) + 2;  
}
```

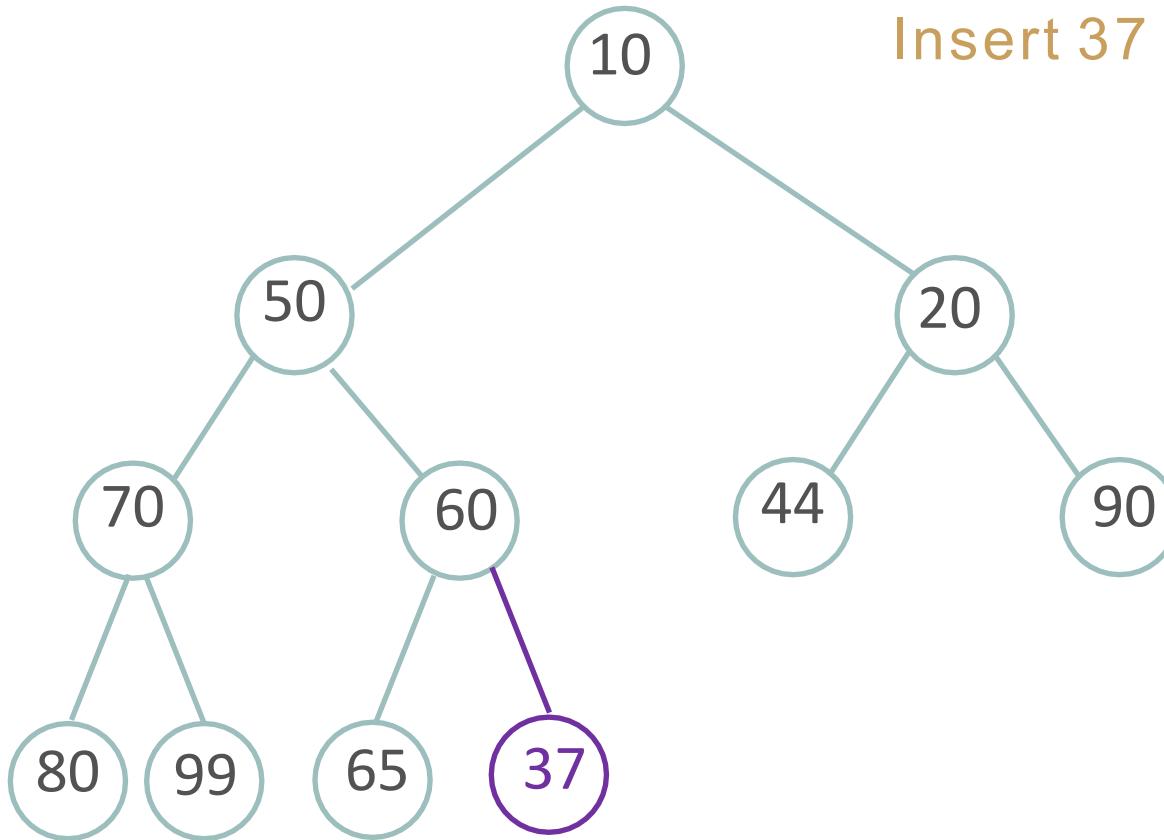
```
int parentIndex(int i) {  
    return (i-1) / 2;  
}
```

# Add



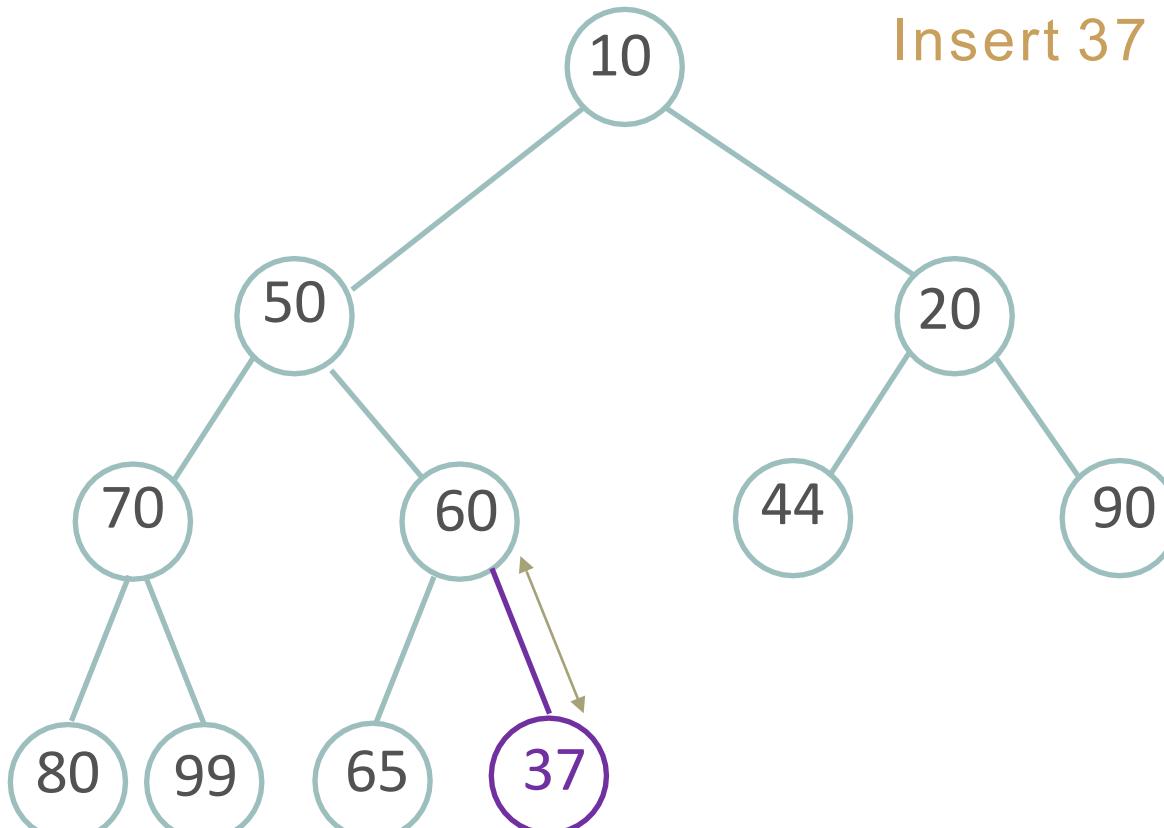
|    |    |    |    |    |    |    |    |    |    |  |  |
|----|----|----|----|----|----|----|----|----|----|--|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 |  |  |
|----|----|----|----|----|----|----|----|----|----|--|--|

# Add



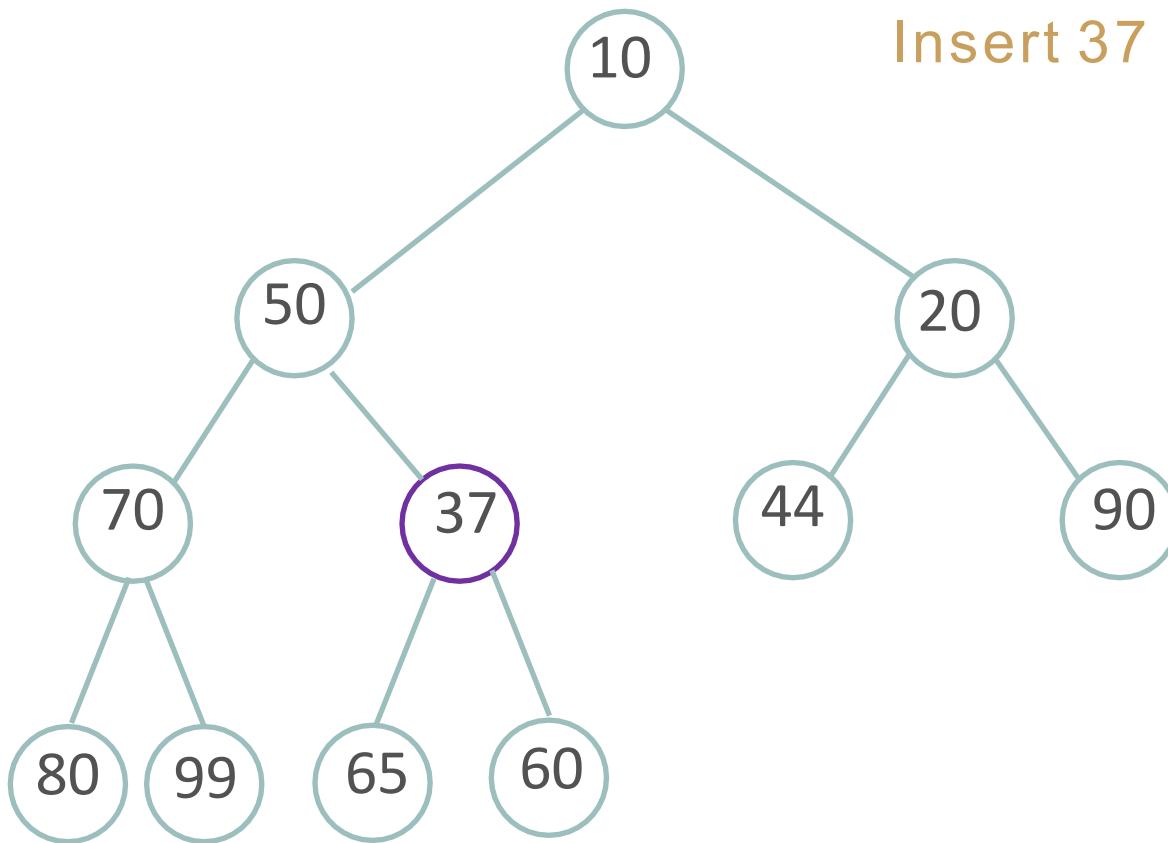
|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 | 37 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 60 | 44 | 90 | 80 | 99 | 65 | 37 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

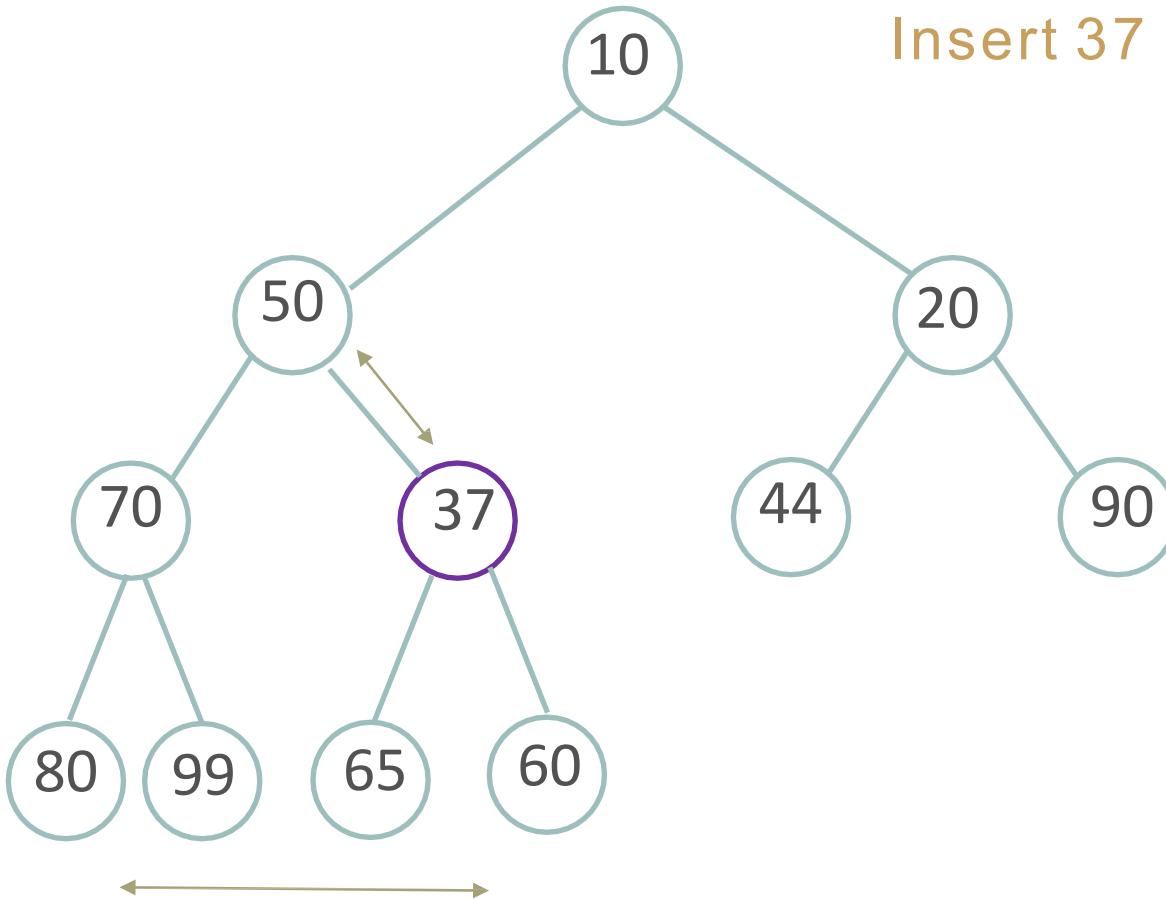
# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 37 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

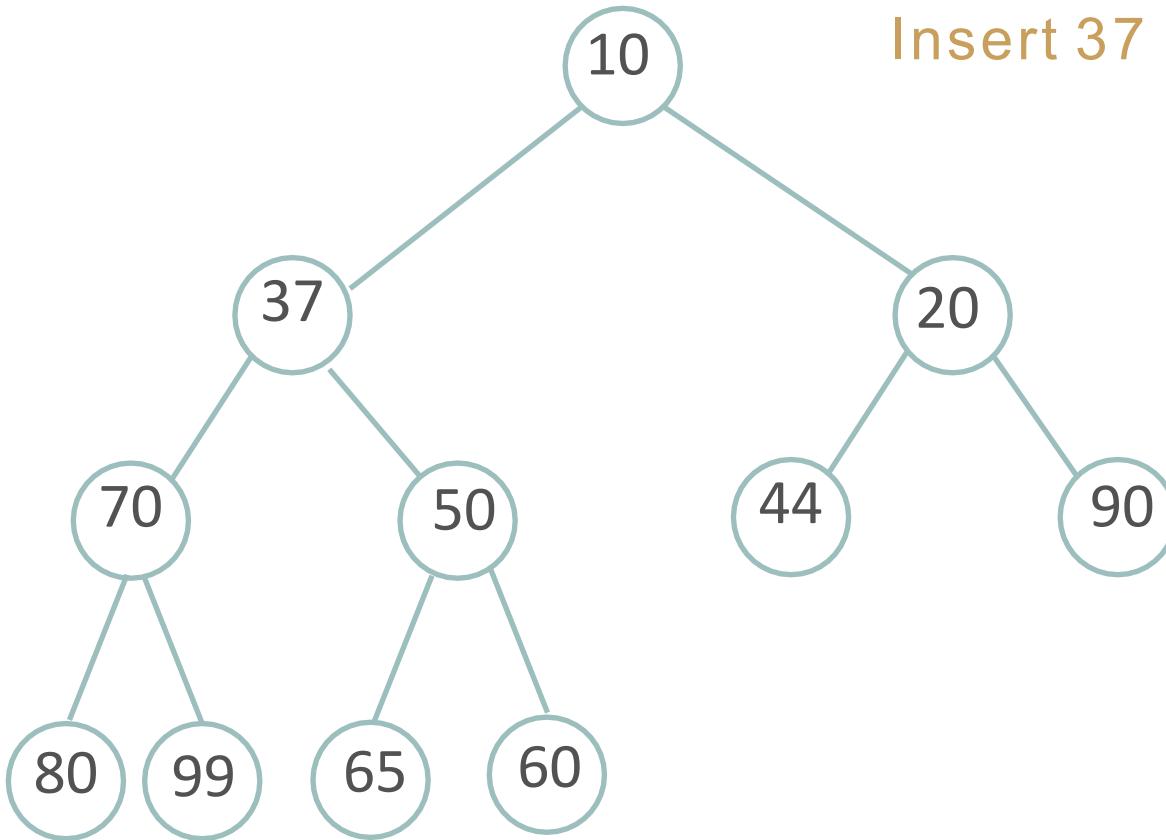
# Add

Insert 37



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 50 | 20 | 70 | 37 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

# Add



|    |    |    |    |    |    |    |    |    |    |    |  |
|----|----|----|----|----|----|----|----|----|----|----|--|
| 10 | 37 | 20 | 70 | 50 | 44 | 90 | 80 | 99 | 65 | 60 |  |
|----|----|----|----|----|----|----|----|----|----|----|--|

# EXAMPLE

Today...

# Generic heaps

- Basic operation for heap

# Generic heaps

- Basic operation for heap
  - comparison
- How to perform comparison for objects?

# Generic heaps

- Basic operation for heap
  - comparison
- How to perform comparison for objects?
  - Use `compareTo` method

# Heap sort

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- How?

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array ☺

# Heap sort

- A simple sorting algorithm that takes advantage of the ordering of heaps to arrange data quickly.
- Add all elements of an array to a priority queue
- Then remove them!
- You have sorted array 😊 → Why?!

# Pseudo code...

A = array to be sorted

H = create new heap

For each element n in A

    add n to H

While (H not empty)

    remove element from H

    add element back into A

# Heap sort complexity

- For sorting an array of size  $N$  heap sort performs:
  - $N$  add
  - $N$  remove

# Heap sort complexity

- For sorting an array of size  $N$  heap sort performs:
  - $N$  add
  - $N$  remove
- Each add or remove is  $\log(N)$

# Heap sort complexity

- For sorting an array of size  $N$  heap sort performs:
  - $N$  add
  - $N$  remove
- Each add or remove is  $\log(N)$

$$\longrightarrow N \log(N)$$

# Heap sort complexity

- For sorting an array of size  $N$  heap sort performs:
  - $N$  add
  - $N$  remove
  - Each add or remove is  $\log(N)$
- Drawback?!

$$\longrightarrow N \log(N)$$

# Heap sort complexity

- For sorting an array of size  $N$  heap sort performs:
  - $N$  add
  - $N$  remove
  - Each add or remove is  $\log(N)$
- Drawback: require extra space

$$\longrightarrow N \log(N)$$

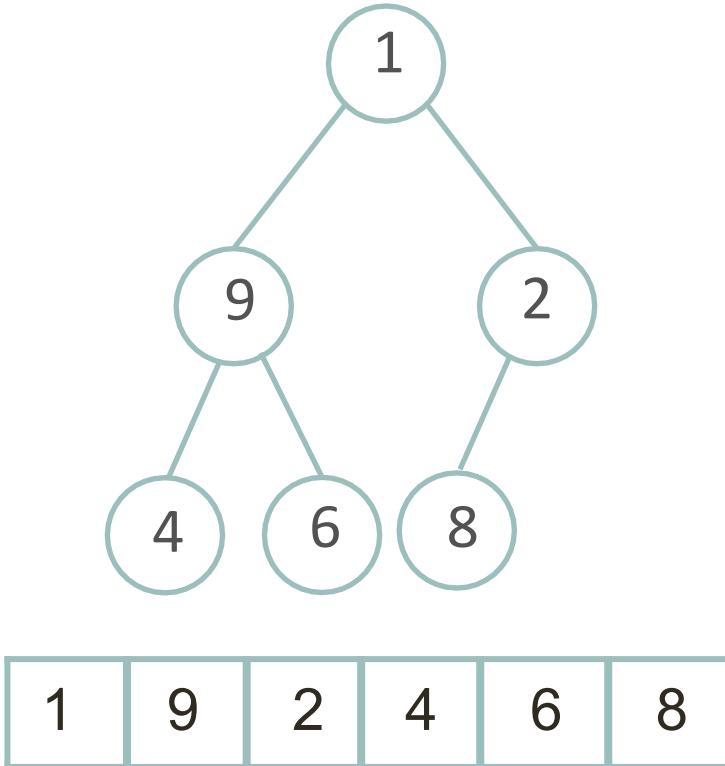
# In-place heap sort

- Use the original array itself as the heap
- The idea is to treat the array as an initially invalid heap and repair it into a proper heap
  - Bubbling elements into their proper position
- Remove elements from the heap and put them at the end of the array.
- min-heap gives you descending order and max-heap gives you ascending order

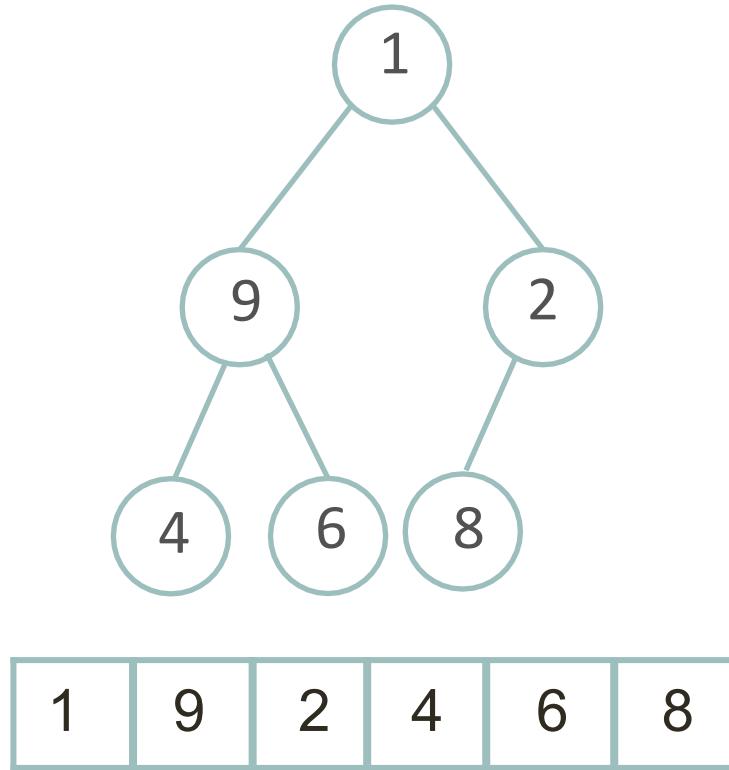
# In-place heap sort

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 9 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|

# In-place heap sort

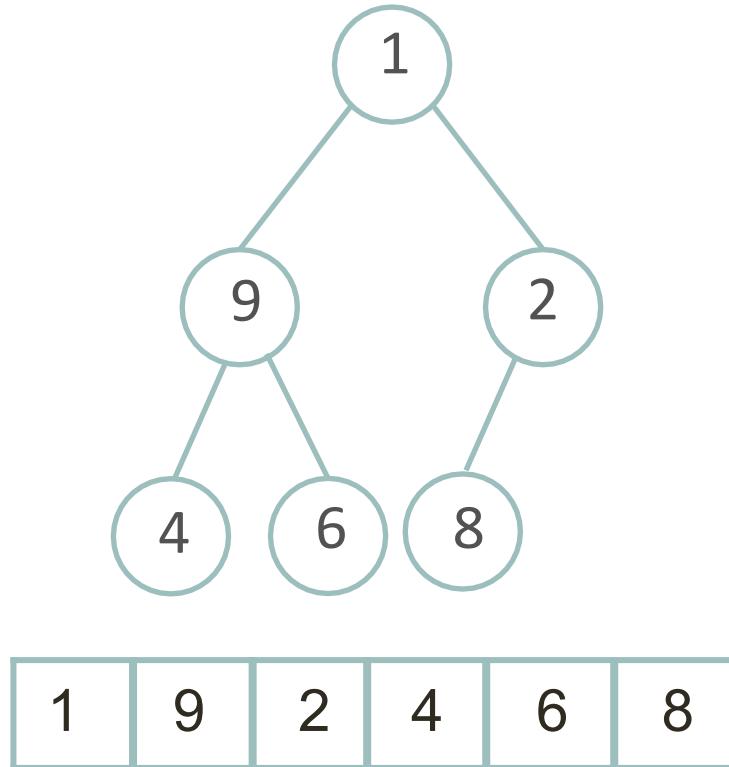


# In-place heap sort



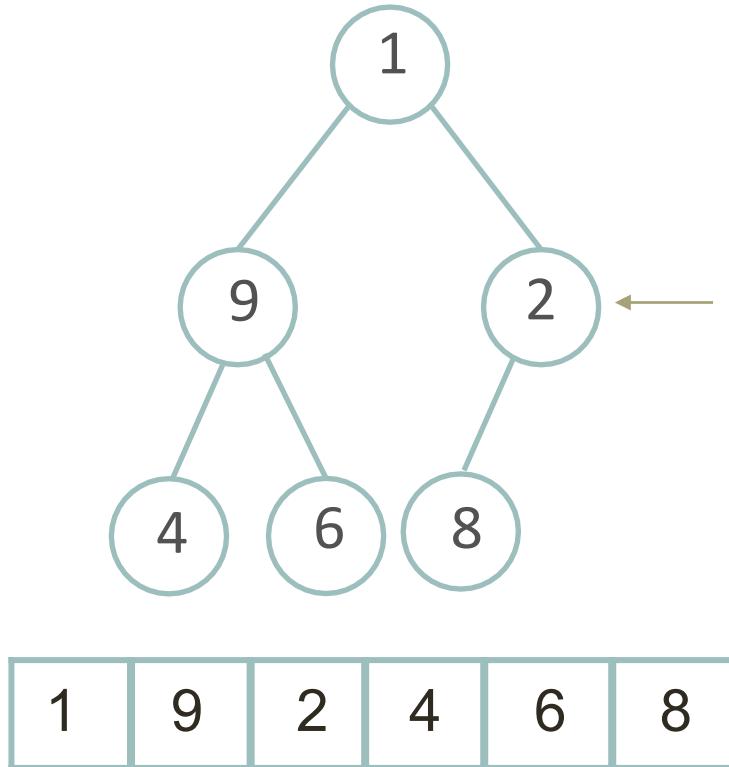
Not a heap → Need to *heapify*

# In-place heap sort



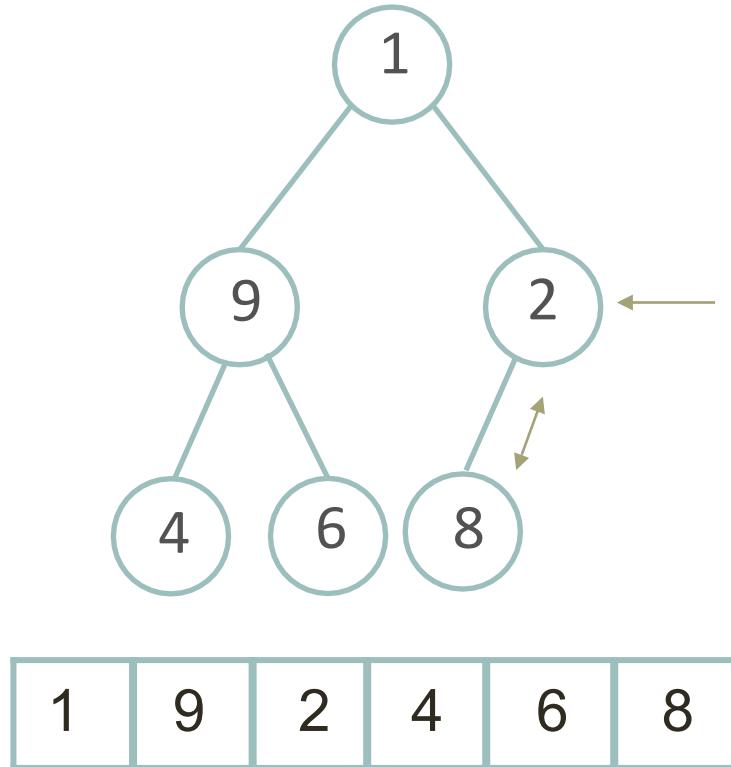
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



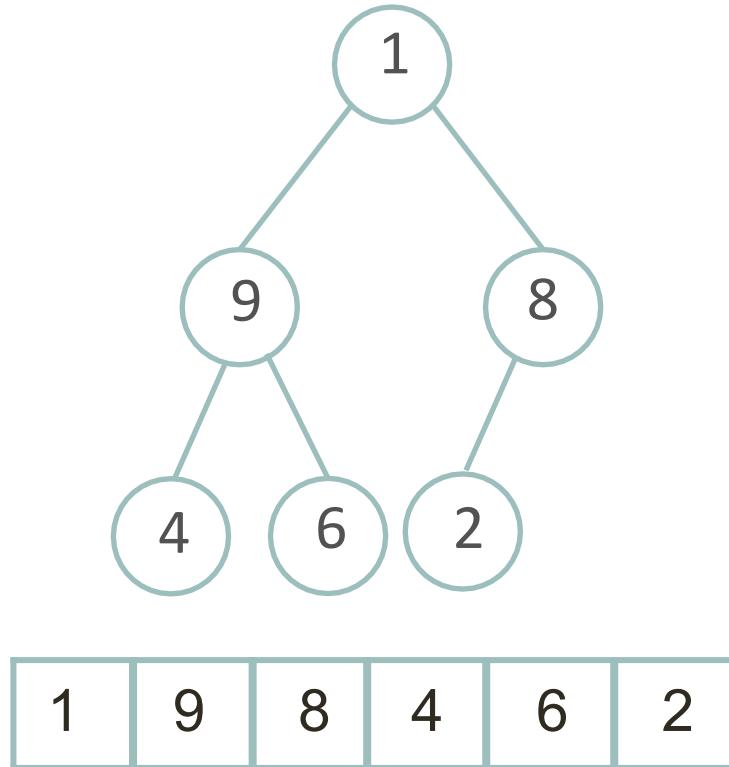
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



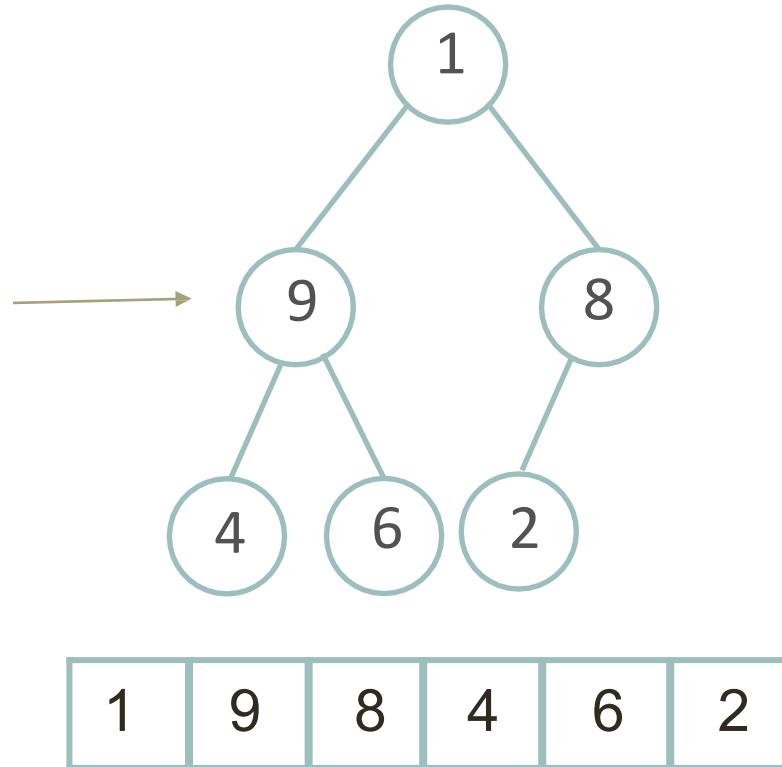
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



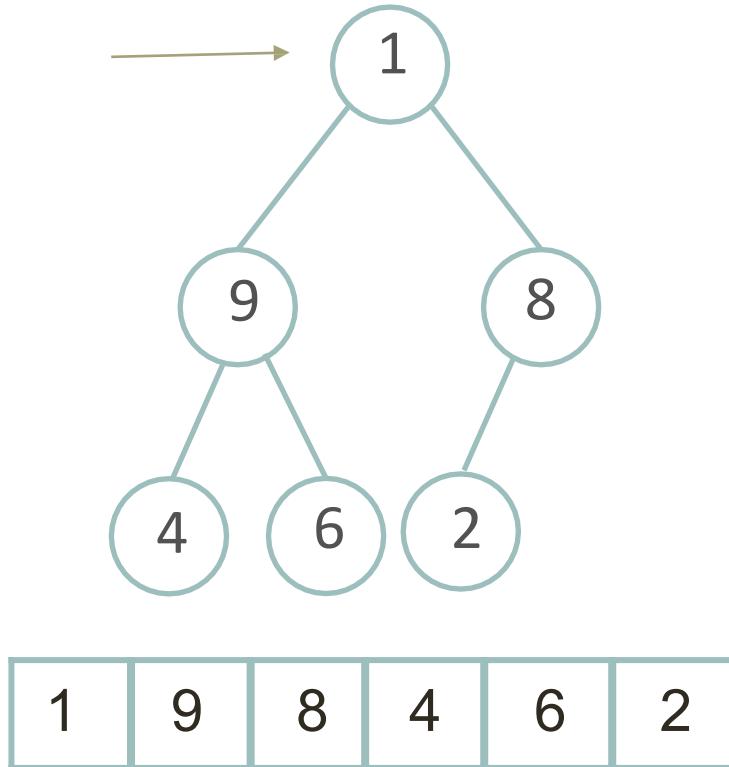
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



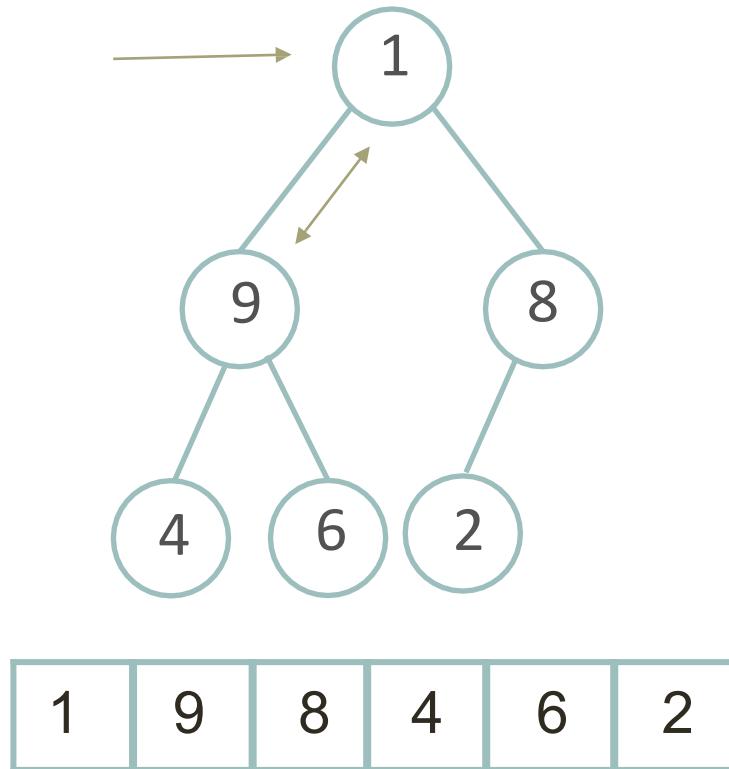
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



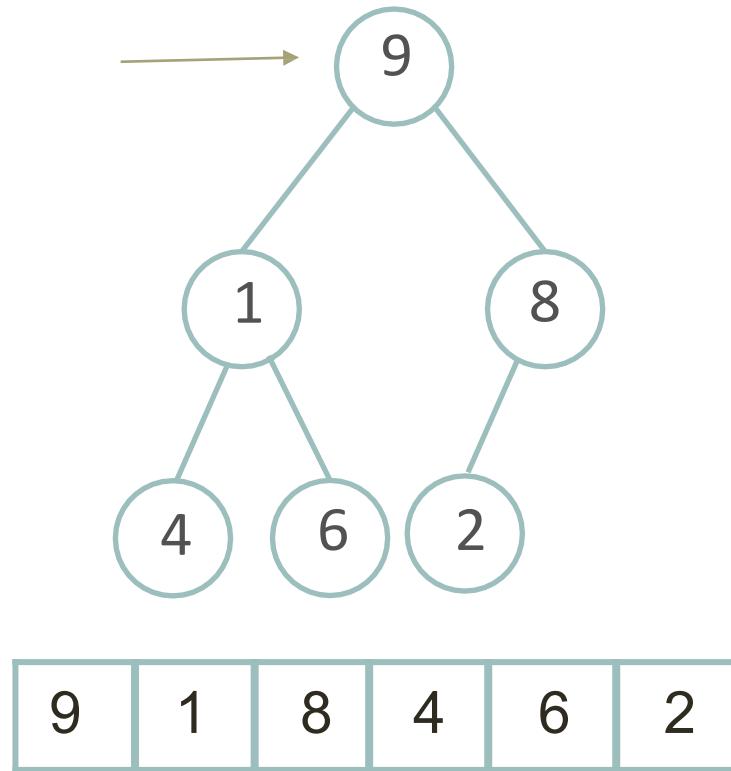
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



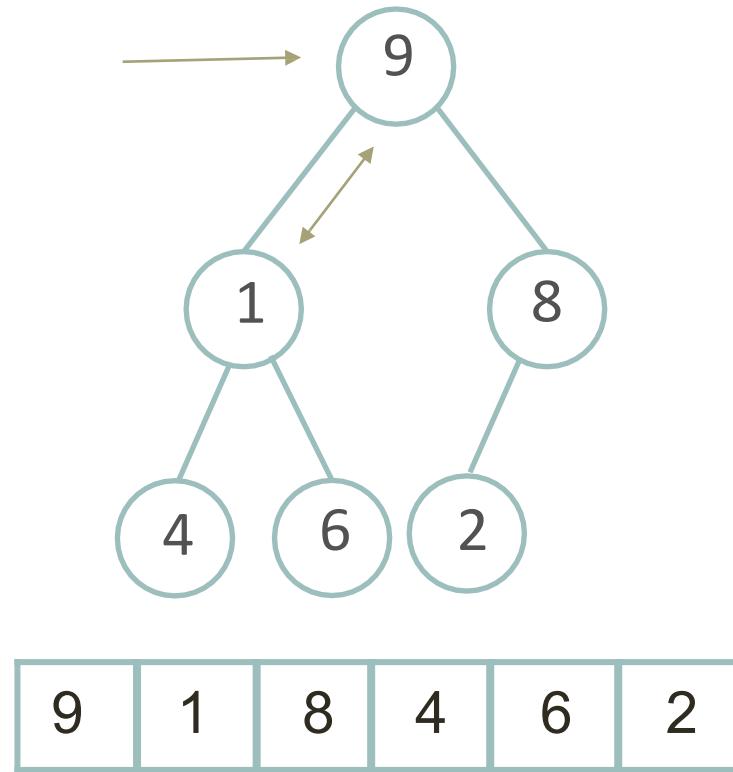
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort



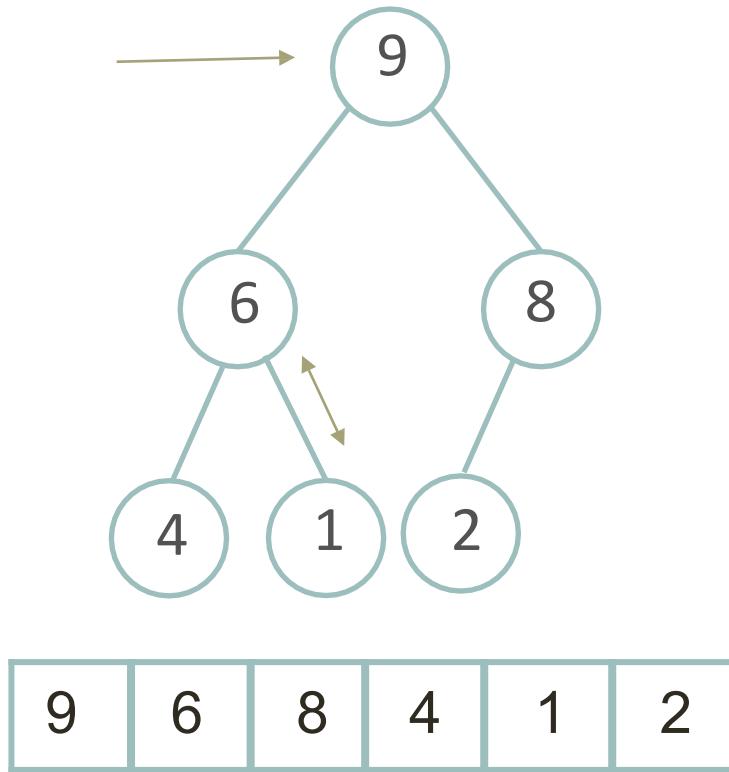
Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

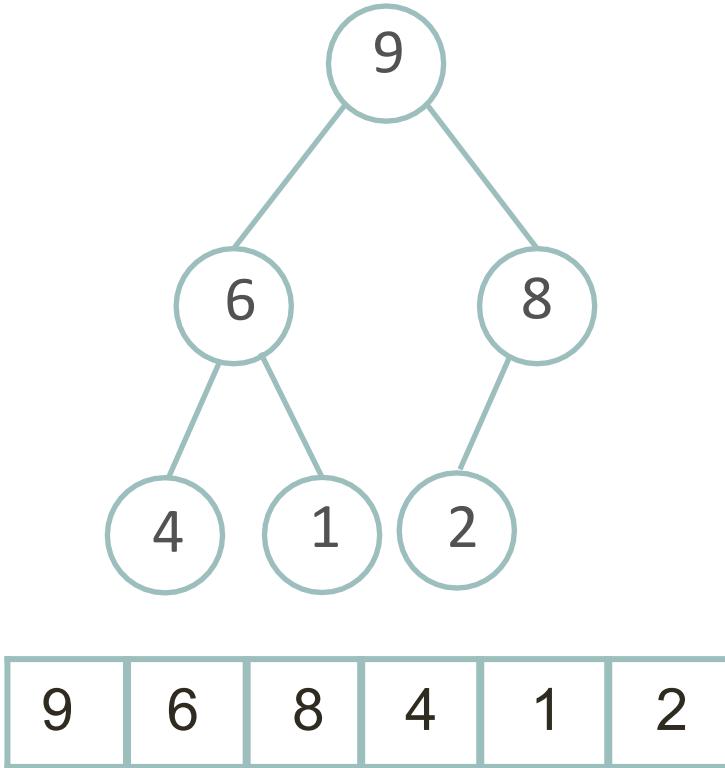


Start at the last parent node → continually sift down the smallest descendant at each level

# In-place heap sort

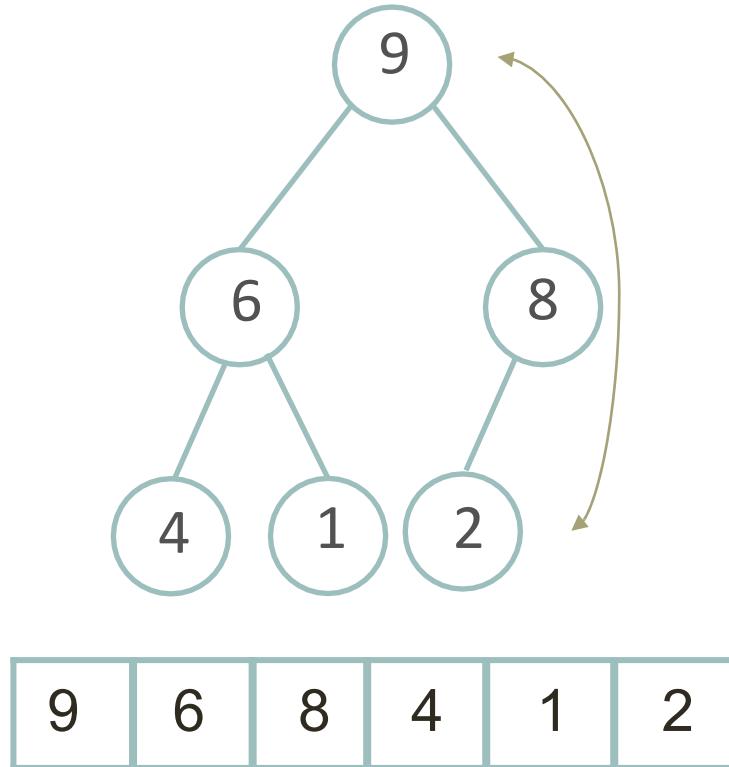


# In-place heap sort



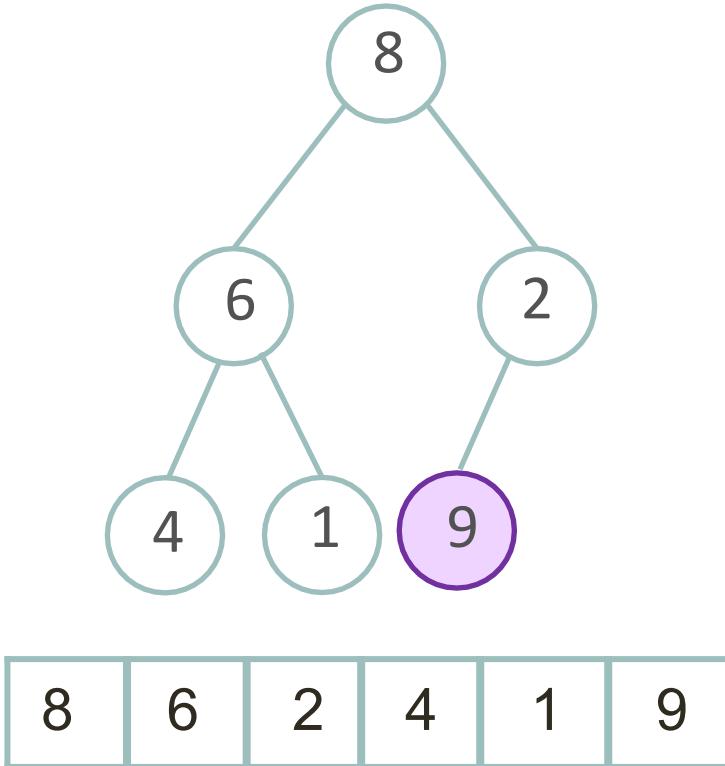
Not yet sorted array!

# In-place heap sort



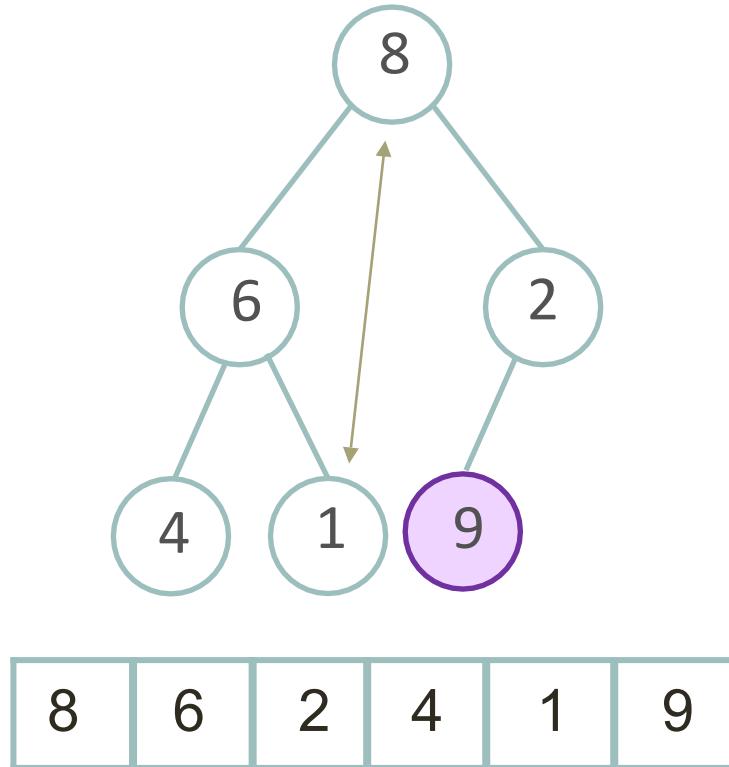
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



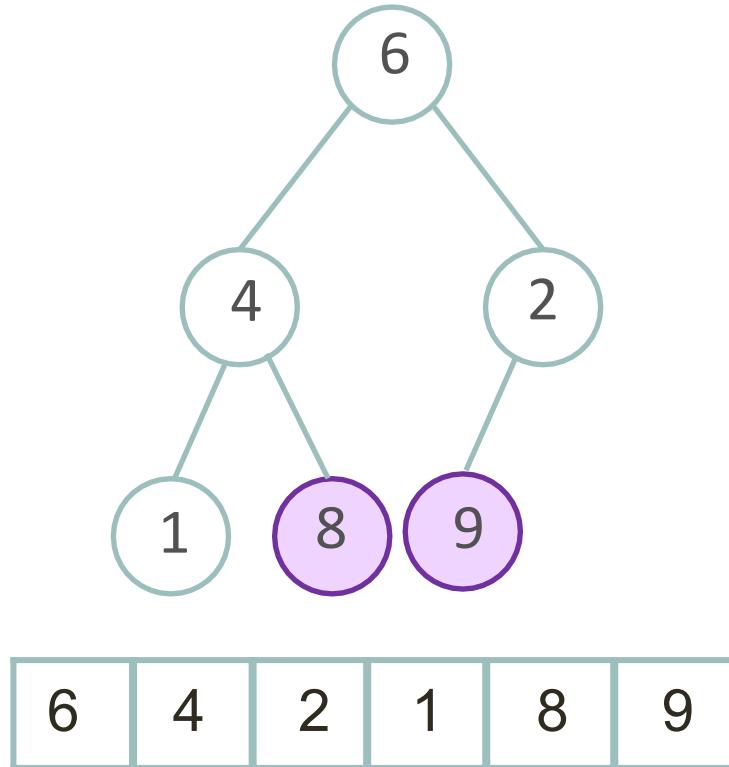
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



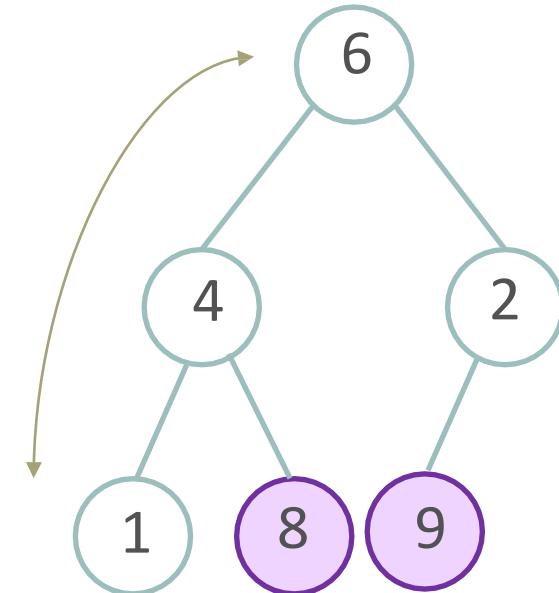
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

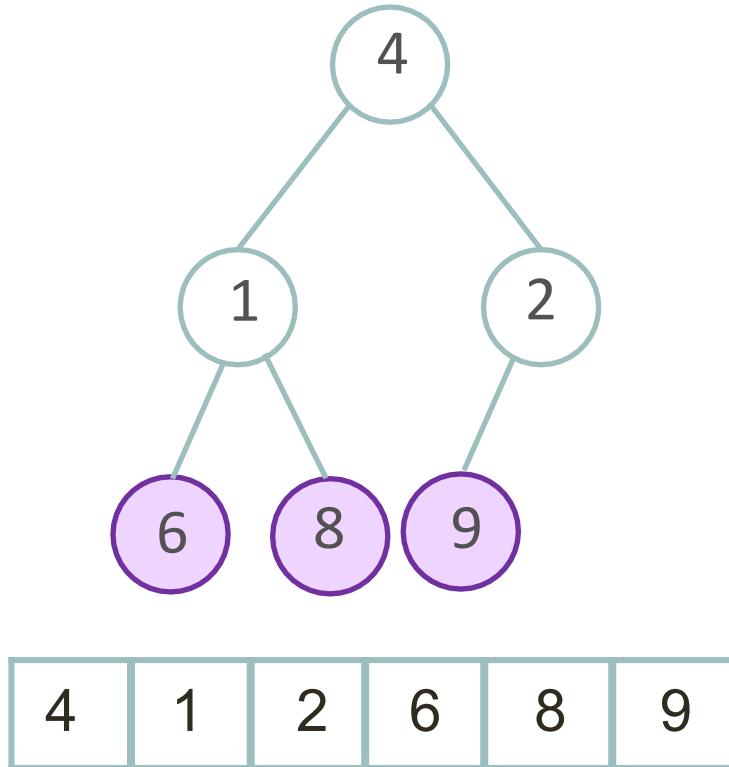
# In-place heap sort



|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 6 | 4 | 2 | 1 | 8 | 9 |
|---|---|---|---|---|---|

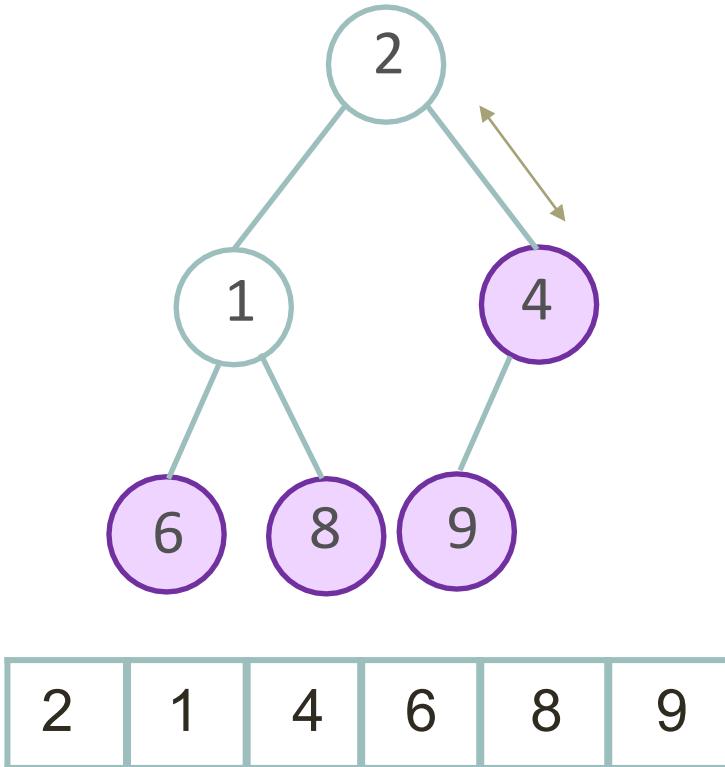
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



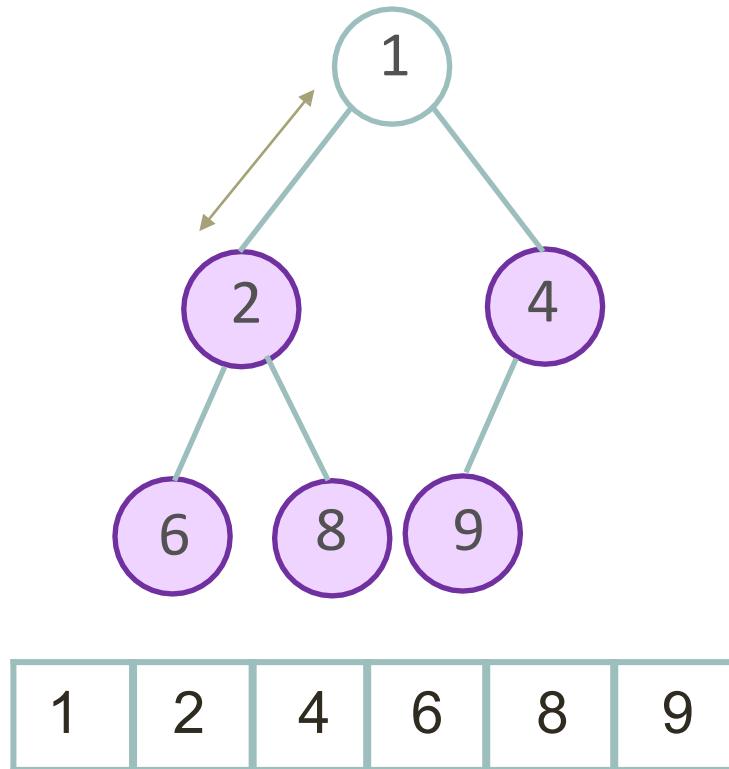
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



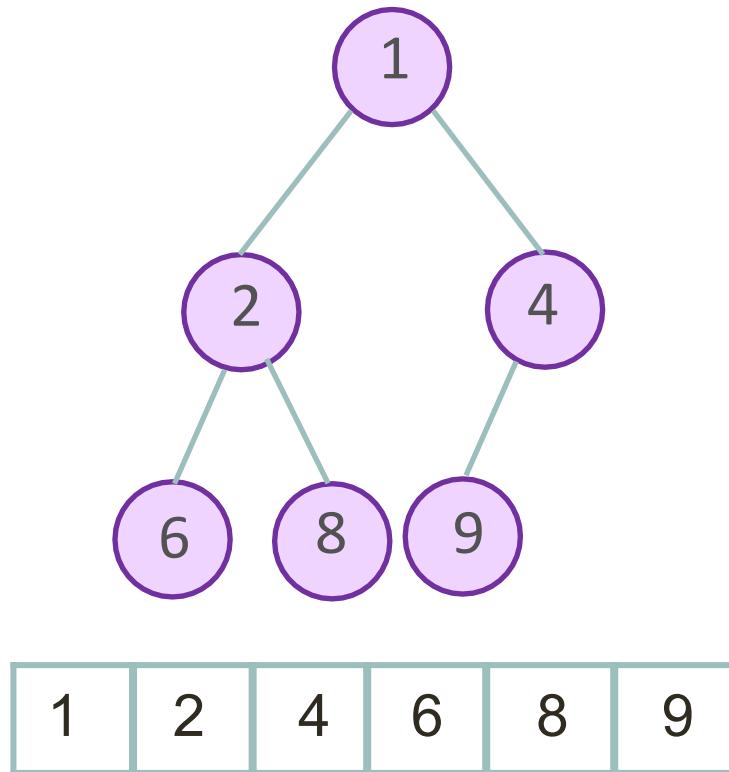
Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

# In-place heap sort



Remove the max, put it at the end  
*Don't forget to sift down if need be*

```
1 // Java program for implementation of Heap Sort
2 public class HeapSort
3 {
4     public void sort(int arr[])
5     {
6         int n = arr.length;
7
8         // Build heap (rearrange array)
9         for (int i = n / 2 - 1; i >= 0; i--)
10            heapify(arr, n, i);
11
12        // One by one extract an element from heap
13        for (int i=n-1; i>=0; i--)
14        {
15            // Move current root to end
16            int temp = arr[0];
17            arr[0] = arr[i];
18            arr[i] = temp;
19
20            // call max heapify on the reduced heap
21            heapify(arr, i, 0);
22        }
23    }
```

```
25 // To heapify a subtree rooted with node i which is
26 // an index in arr[]. n is size of heap
27 void heapify(int arr[], int n, int i)
28 {
29     int largest = i; // Initialize largest as root
30     int l = 2*i + 1; // left = 2*i + 1
31     int r = 2*i + 2; // right = 2*i + 2
32
33     // If left child is larger than root
34     if (l < n && arr[l] > arr[largest])
35         largest = l;
36
37     // If right child is larger than largest so far
38     if (r < n && arr[r] > arr[largest])
39         largest = r;
40
41     // If largest is not root
42     if (largest != i)
43     {
44         int swap = arr[i];
45         arr[i] = arr[largest];
46         arr[largest] = swap;
47
48         // Recursively heapify the affected sub-tree
49         heapify(arr, n, largest);
50     }
51 }
52
53 /* A utility function to print array of size n */
54 static void printArray(int arr[])
55 {
56     int n = arr.length;
57     for (int i=0; i<n; ++i)
58         System.out.print(arr[i]+ " ");
59     System.out.println();
60 }
```

```
61
62 // Driver program
63 public static void main(String args[])
64 {
65     int arr[] = {12, 11, 13, 5, 6, 7};
66     int n = arr.length;
67
68     HeapSort ob = new HeapSort();
69     ob.sort(arr);
70
71     System.out.println("Sorted array is");
72     printArray(arr);
73 }
74 }
75 }
```

[https://www.youtube.com/watch?v=2DmK\\_H7ldTo](https://www.youtube.com/watch?v=2DmK_H7ldTo)

# Implementation

You'll do for your assignment this week!

Next time...

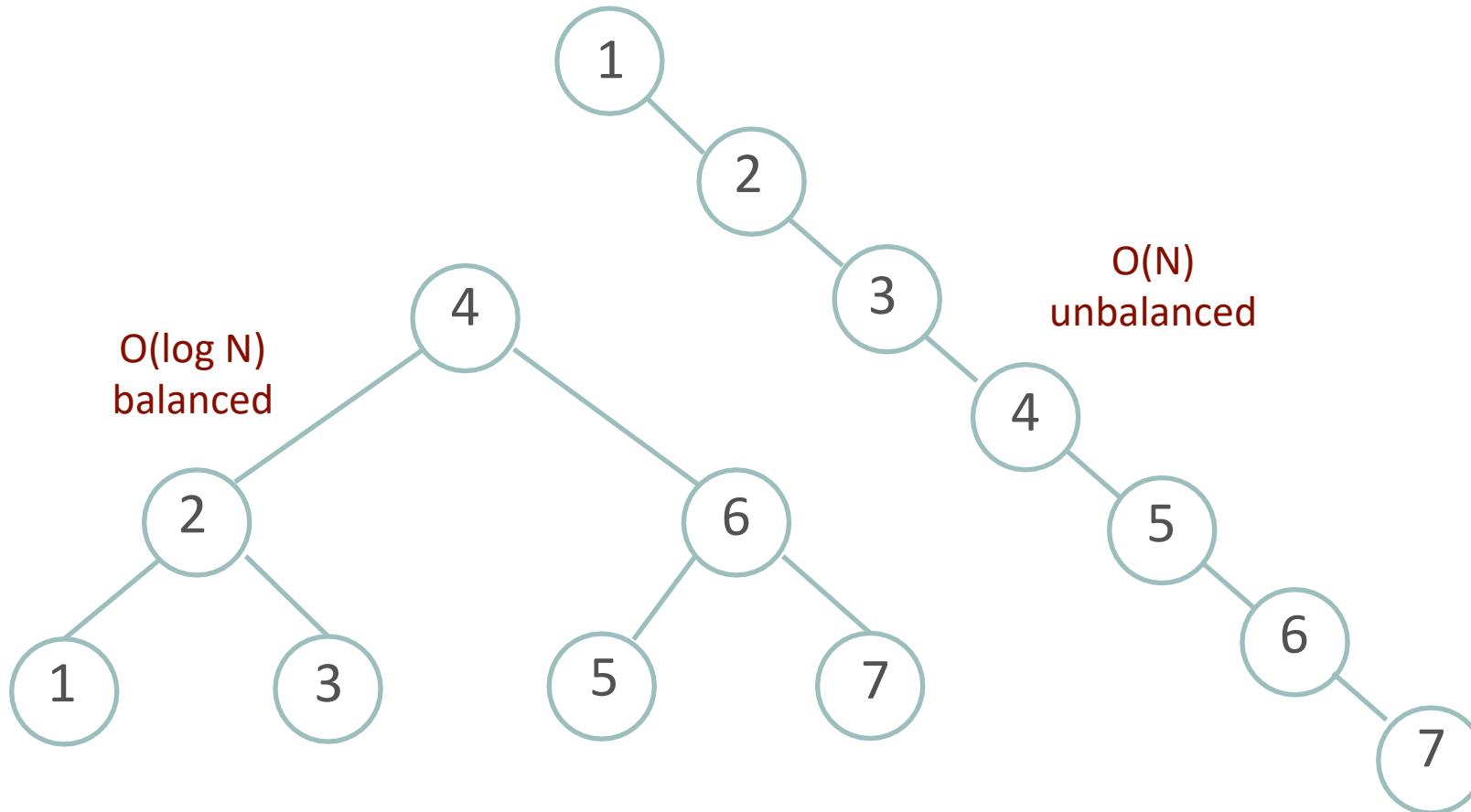
# Hashing

# Scapegoat Tree

CSC220 | Computer Science | University of Miami

# Binary search tree: balanced vs unbalanced

All operations depend on how well-balanced the tree is



- ▶ Scapegoat trees keep two counters:
  - ▶  $n$ : the number of items in the tree
  - ▶  $q$ : an overestimate of  $n$
- ▶ We maintain the invariants:
  - ▶  $q/2 \leq n \leq q$
  - ▶ No node has depth greater than  $\log_{3/2} q$

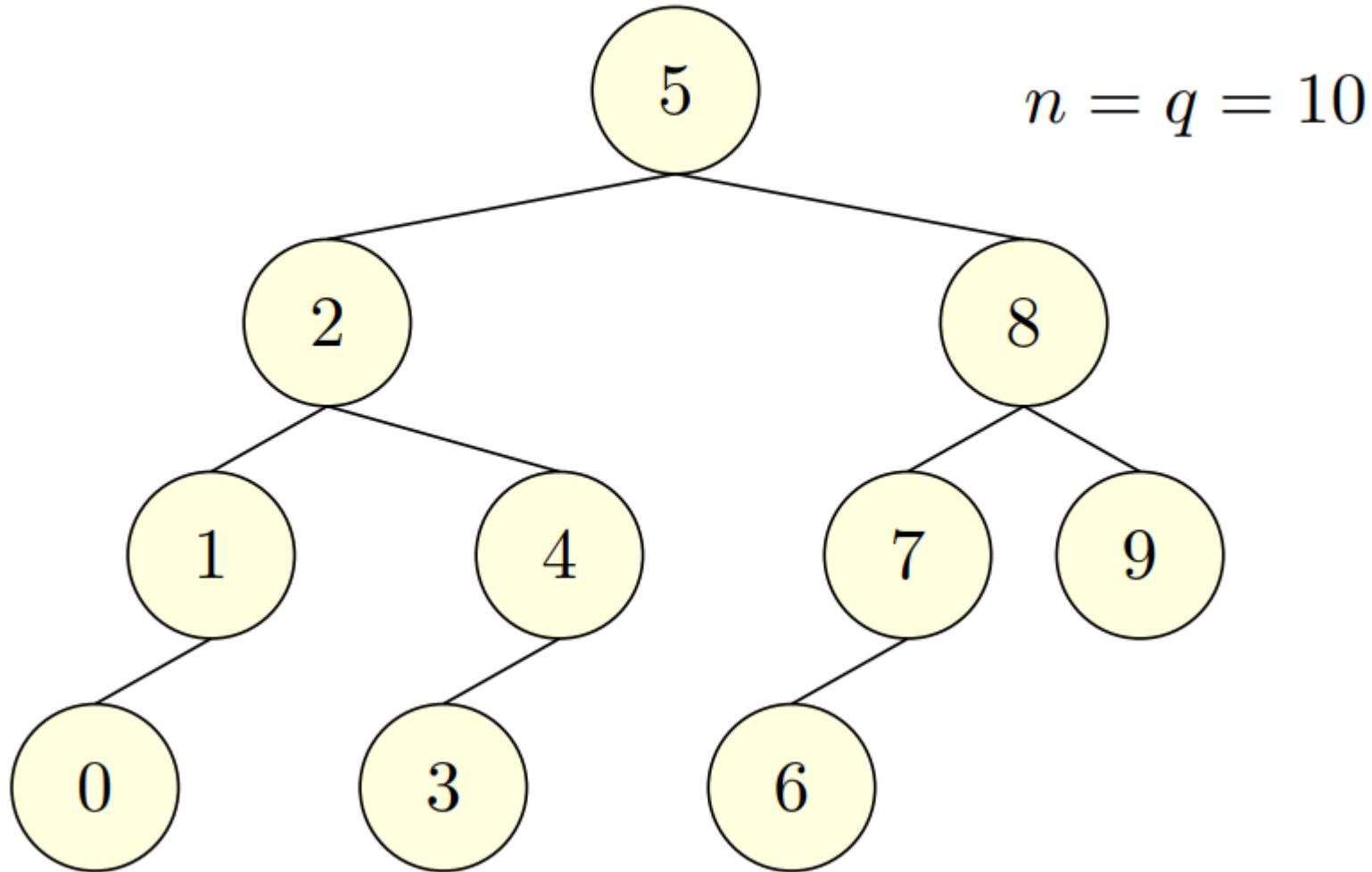
- ▶ To remove the value  $x$  from a ScapegoatTree:
  1. run the standard deletion algorithm for binary search trees
  2. decrement  $n$
  3. if  $n < q/2$  then
    - ▶ rebuild the entire tree and set  $q=n$

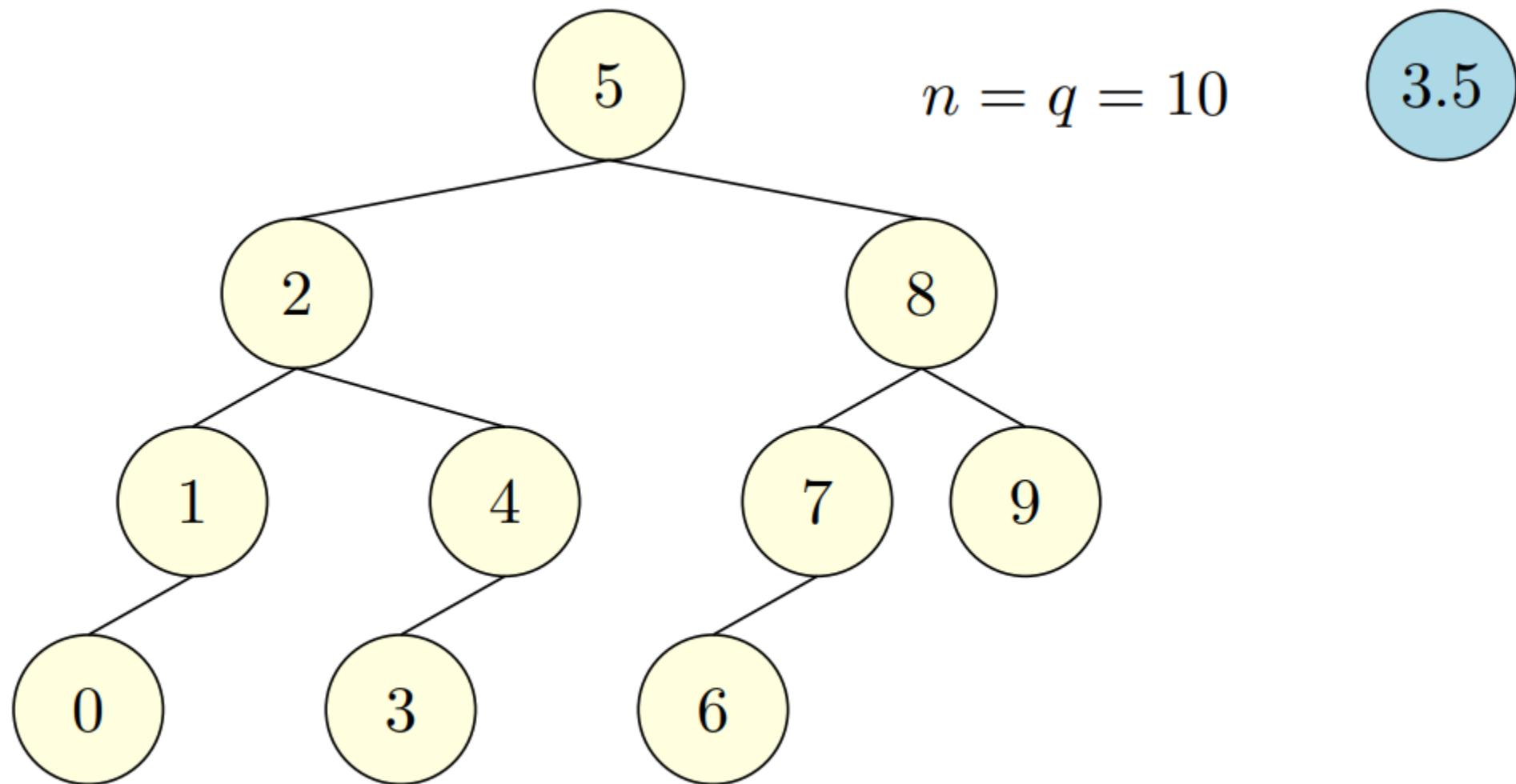
- ▶ To insert the value  $x$  into a ScapegoatTree:
  - ▶ create a node  $u$  and insert in the normal way
  - ▶ increment  $n$  and  $q$
  - ▶ If the depth of  $u$  is greater than  $\log_{3/2} q$ , then
    - ▶ walk up to the root from  $u$  until reaching a node  $w$  with

$$\text{size}(w) > \frac{2}{3} \text{size}(w.\text{parent})$$

- ▶ Rebuild the subtree rooted at  $w.\text{parent}$

Inserting into a scapegoat tree (good case)

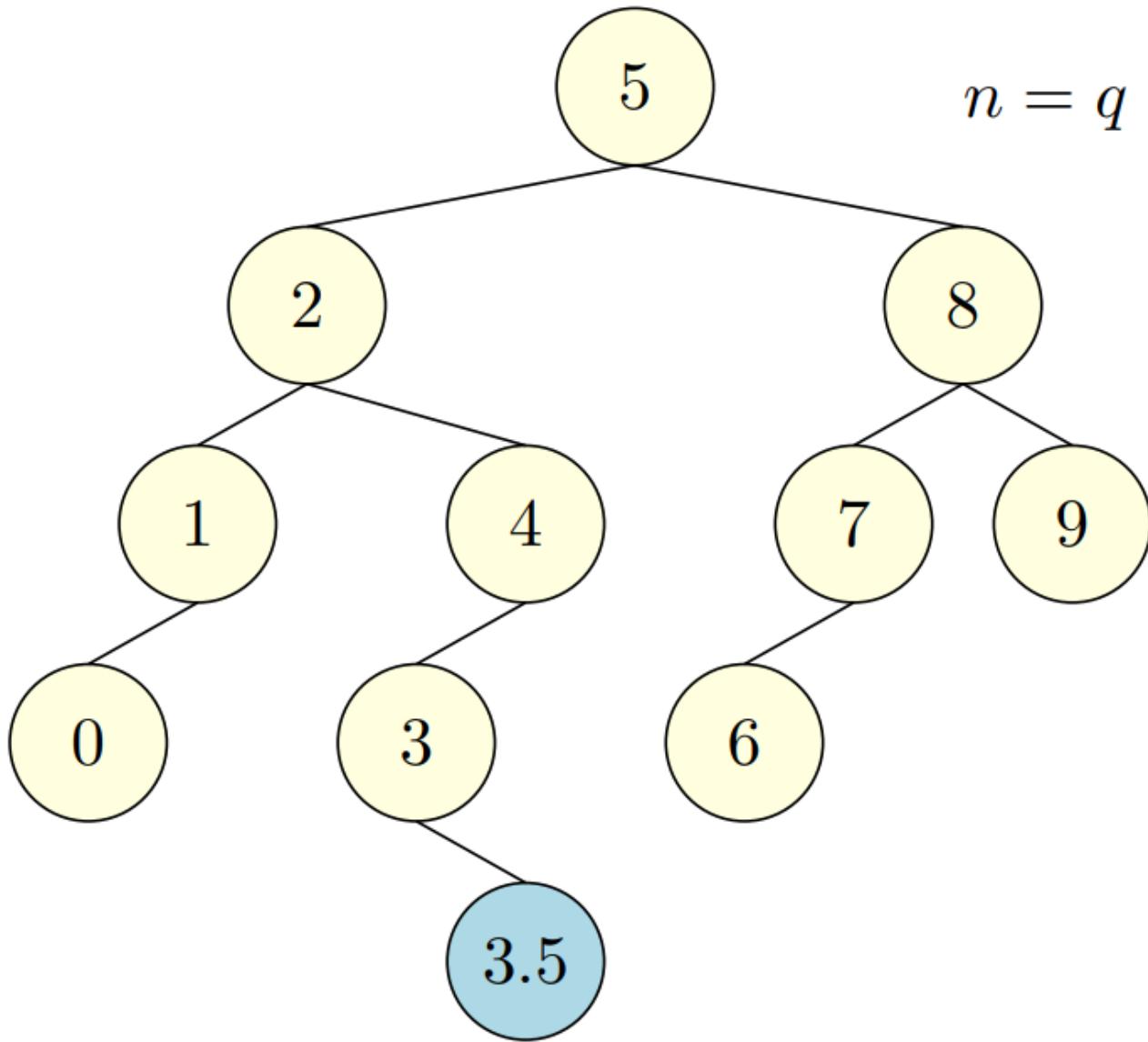




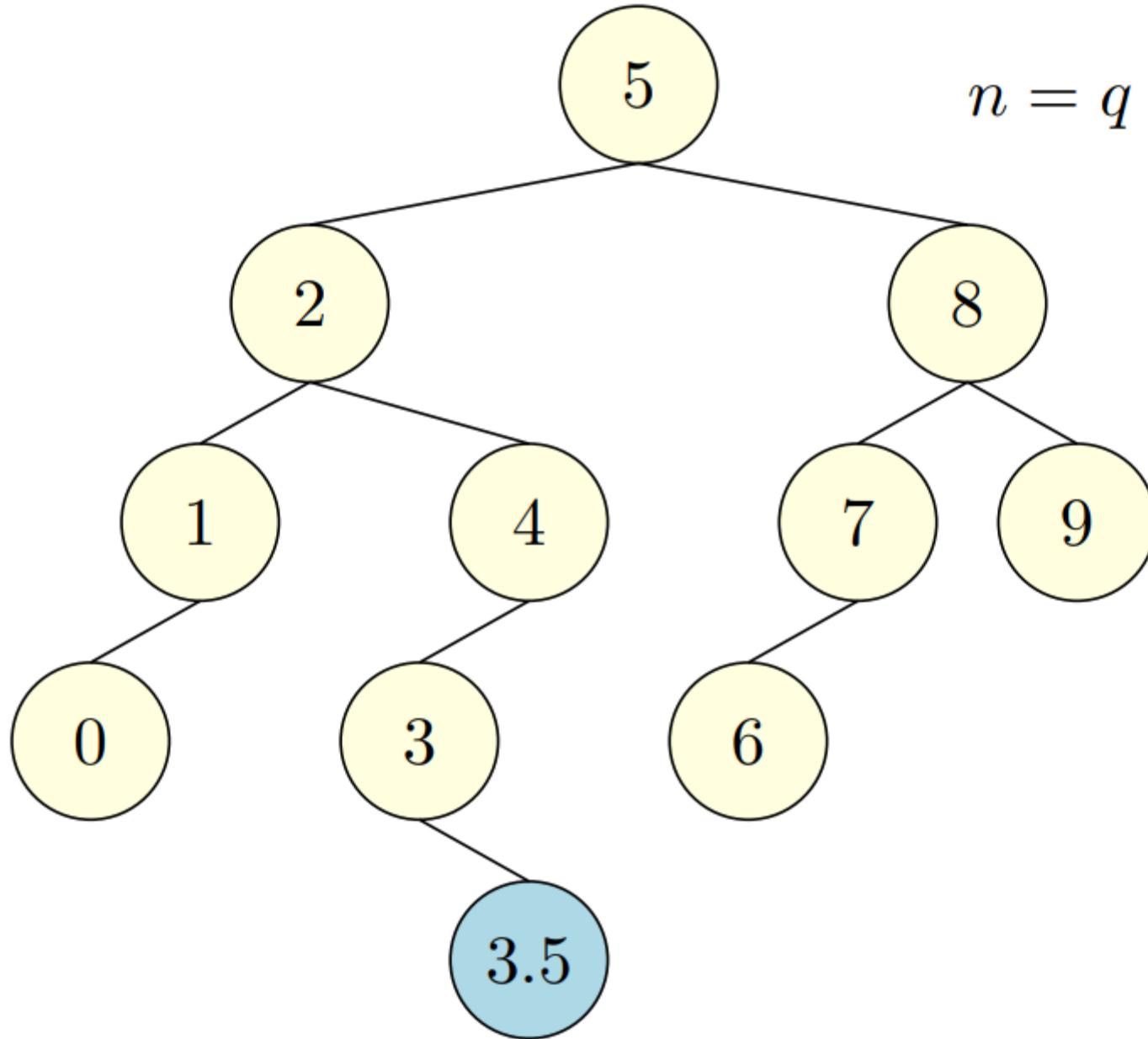
$$n = q = 10$$

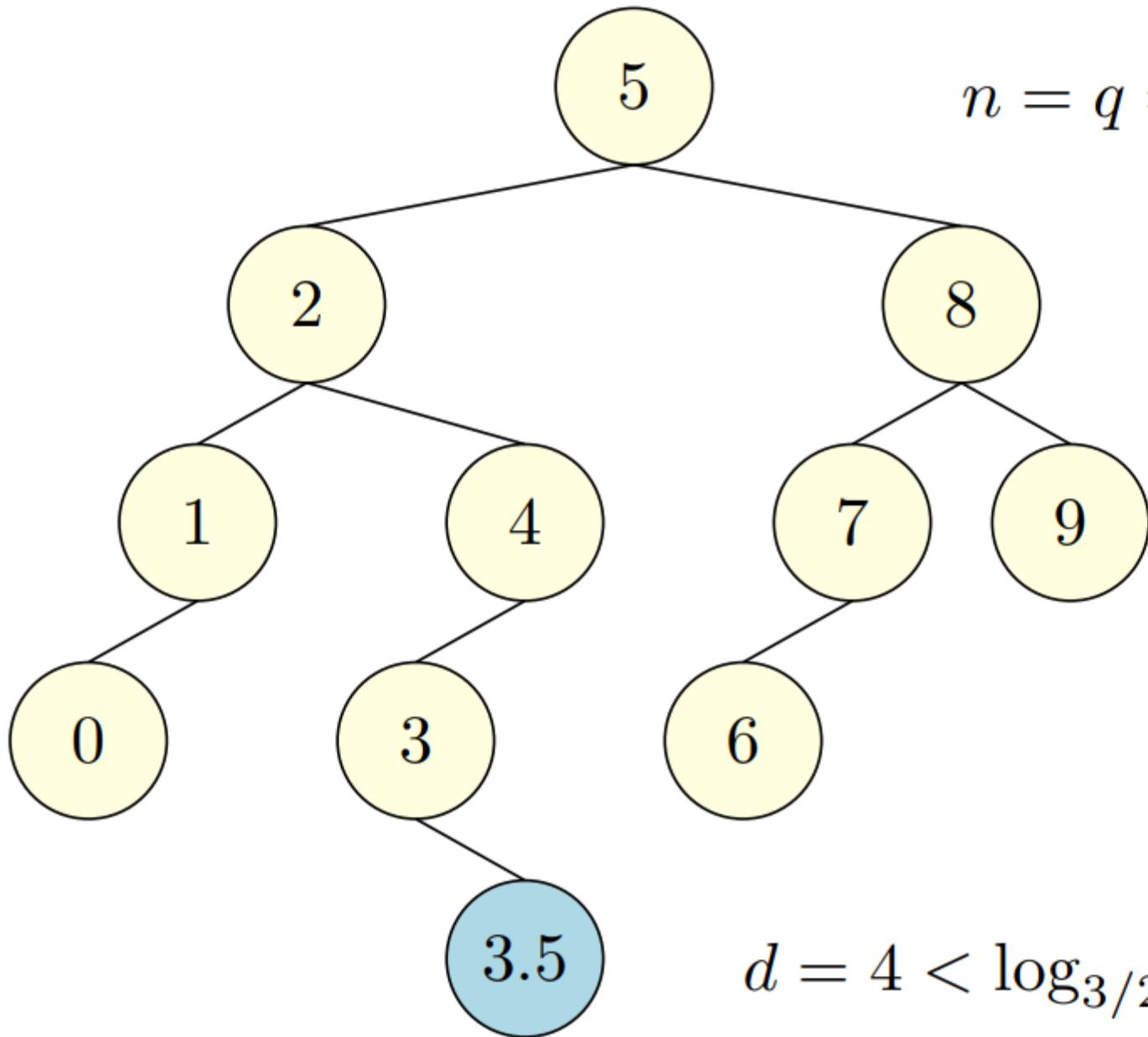
3.5

$$n = q = 10$$



$$n = q = 11$$

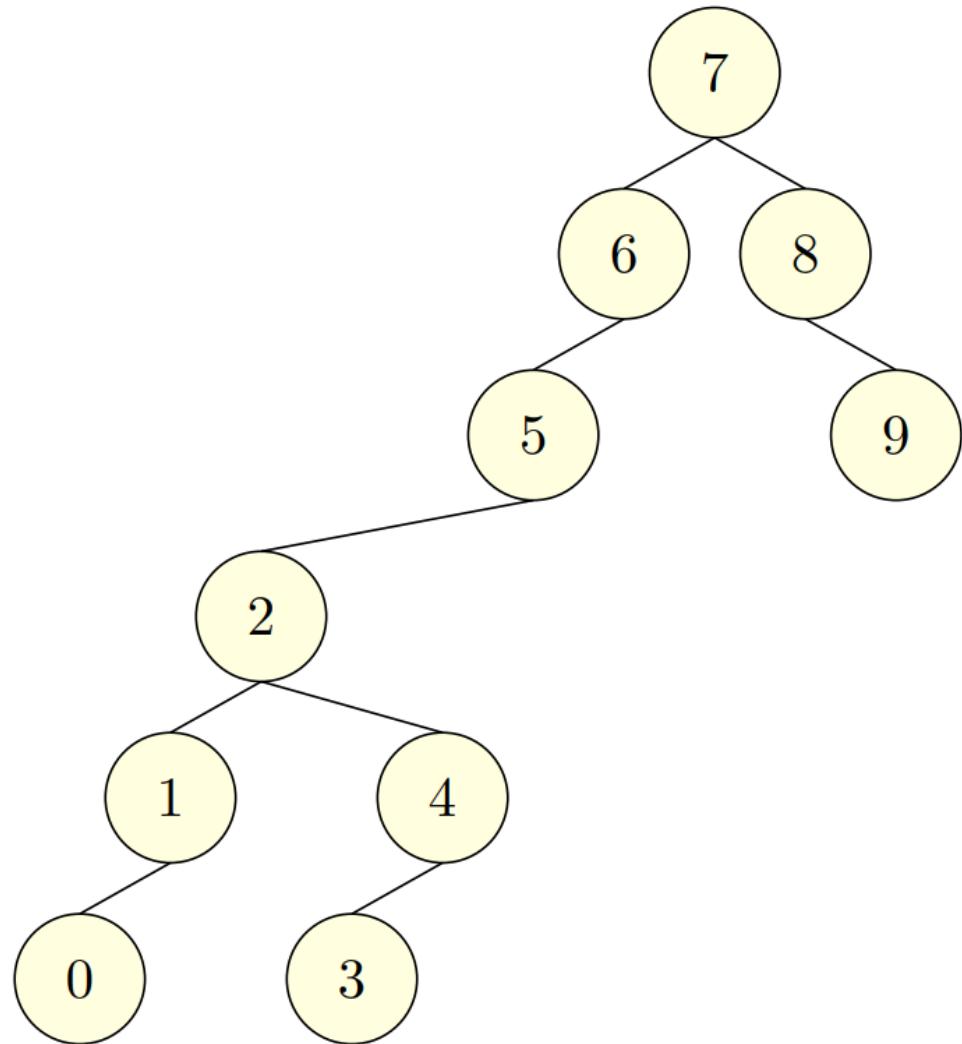


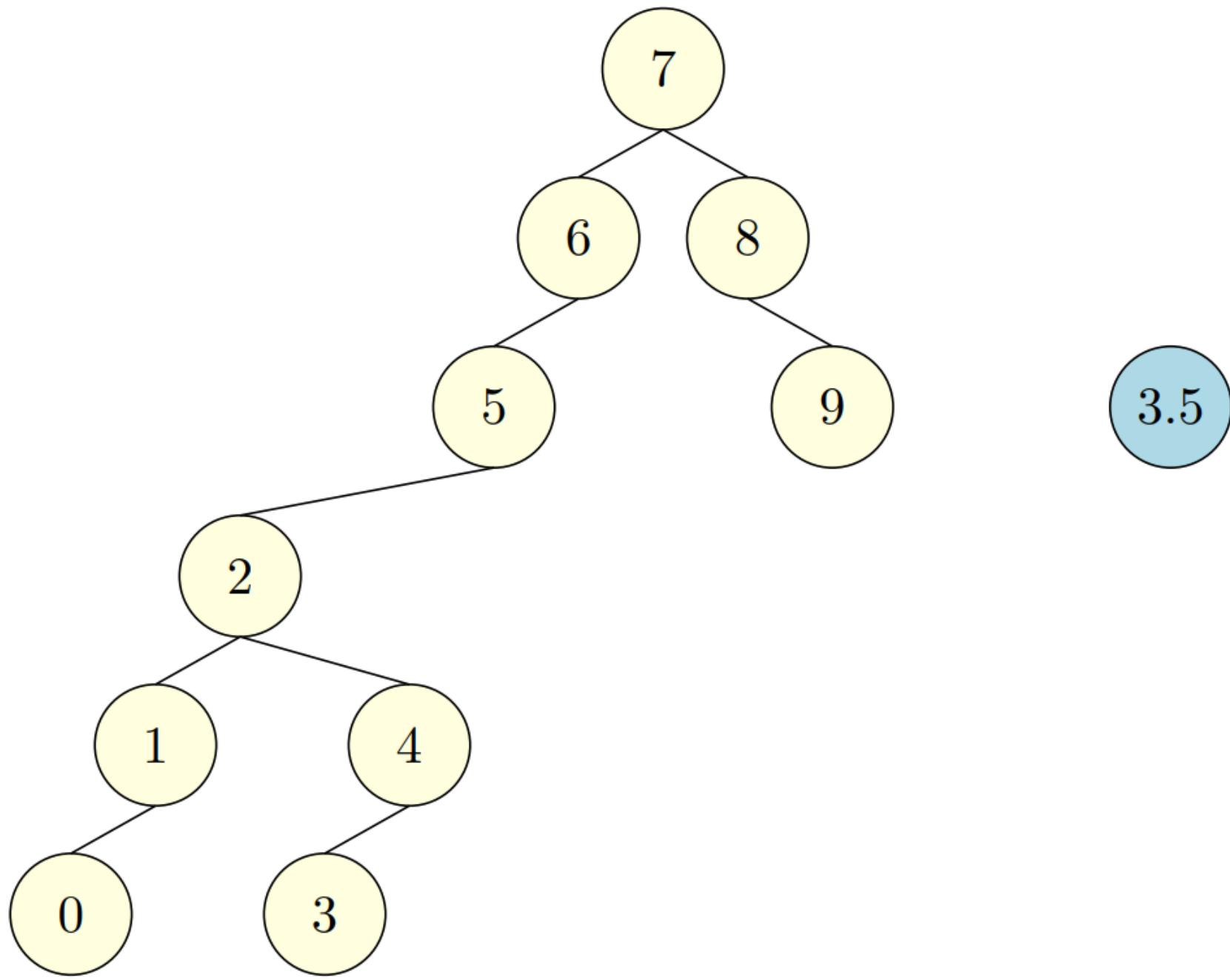


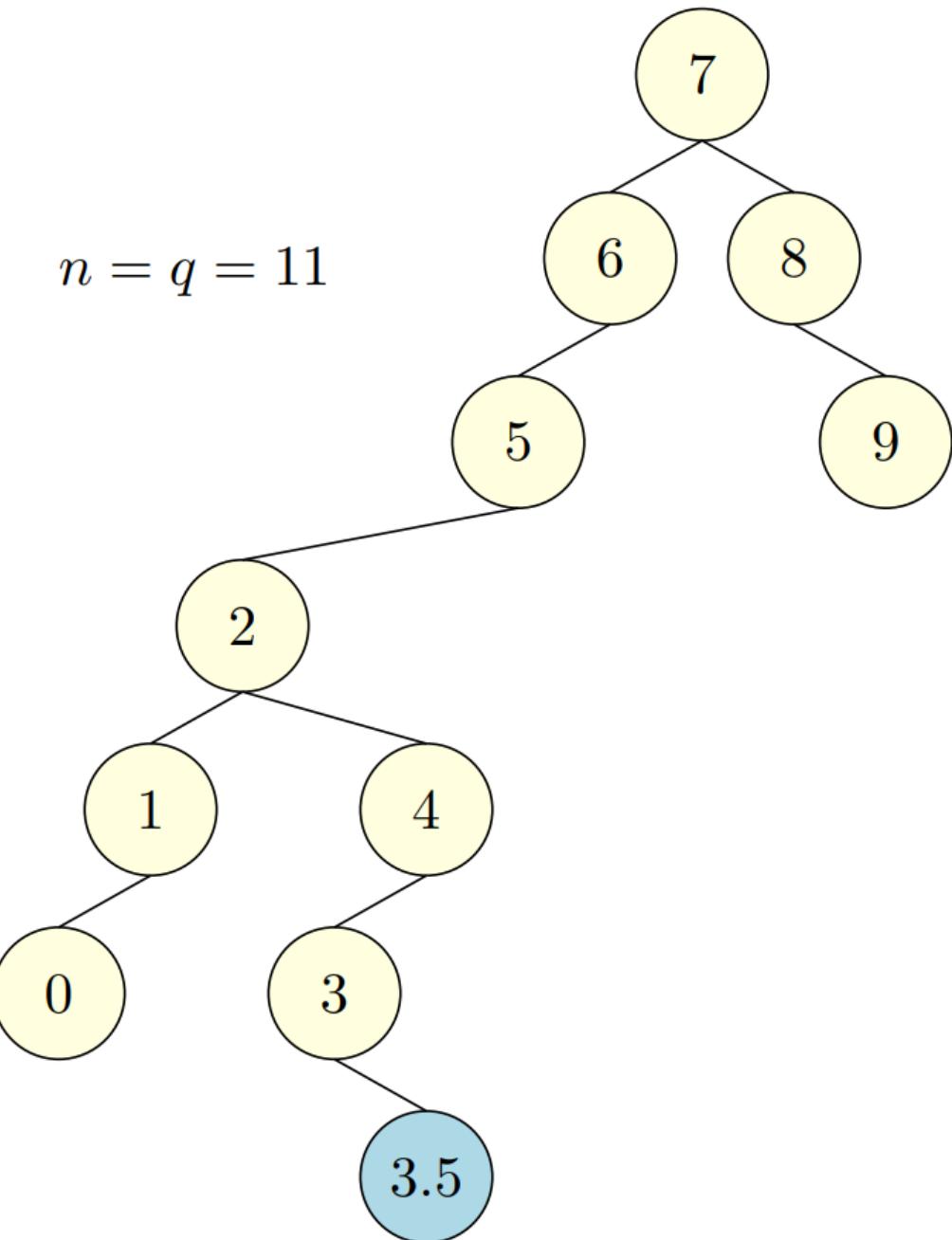
$$n = q = 11$$

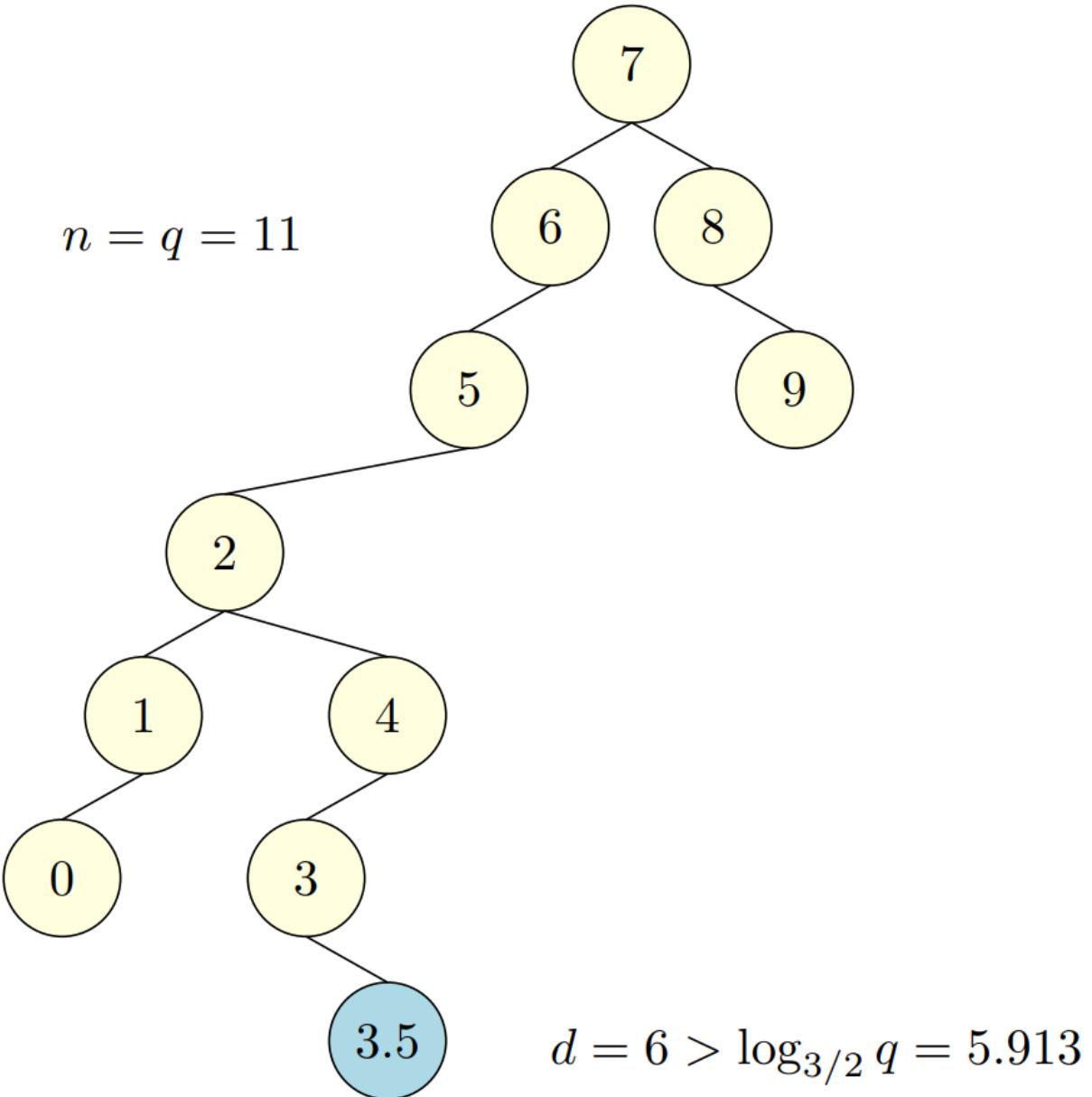
$$d = 4 < \log_{3/2} q \approx 5.913$$

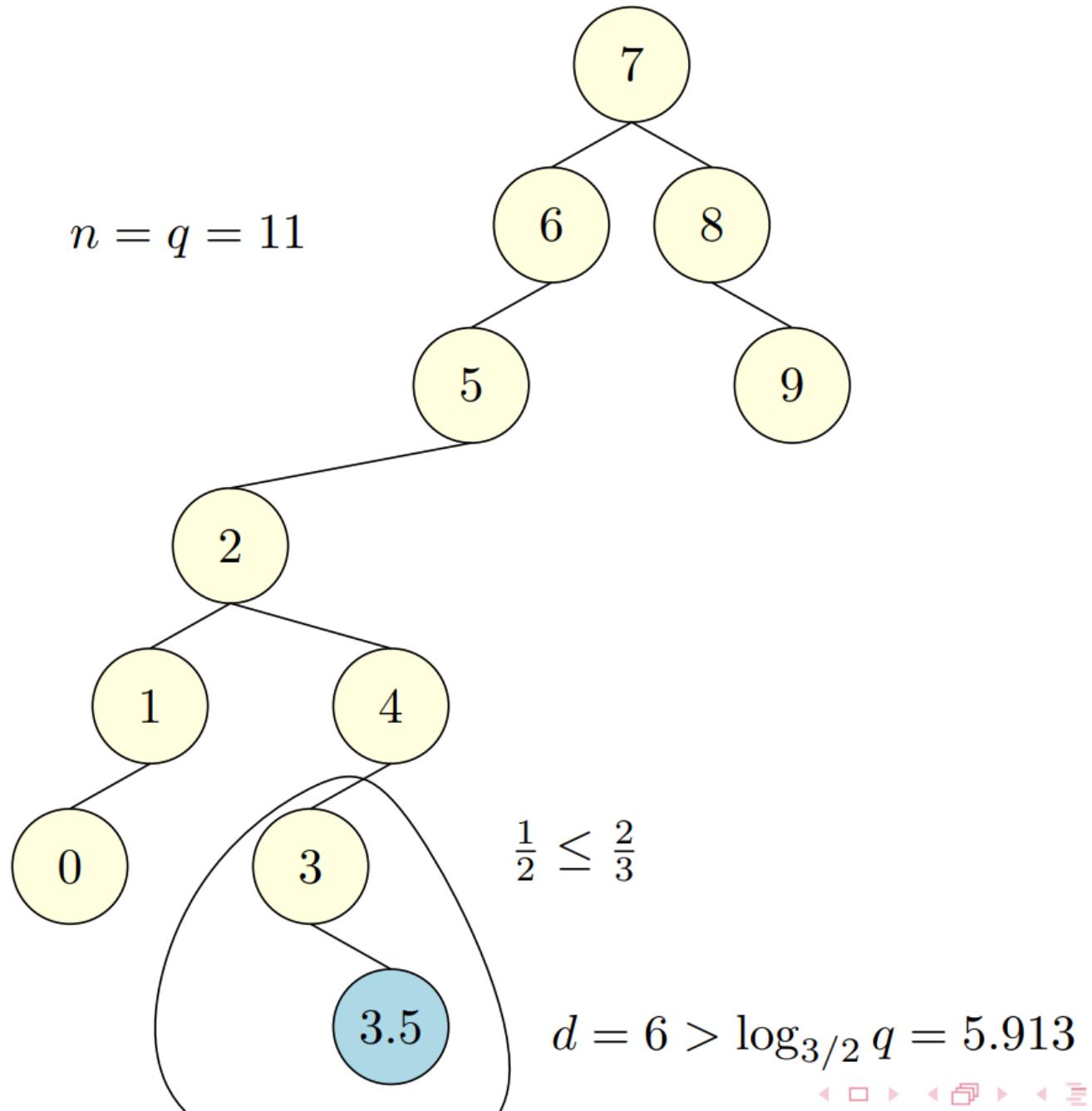
## Inserting into a scapegoat tree (bad case)

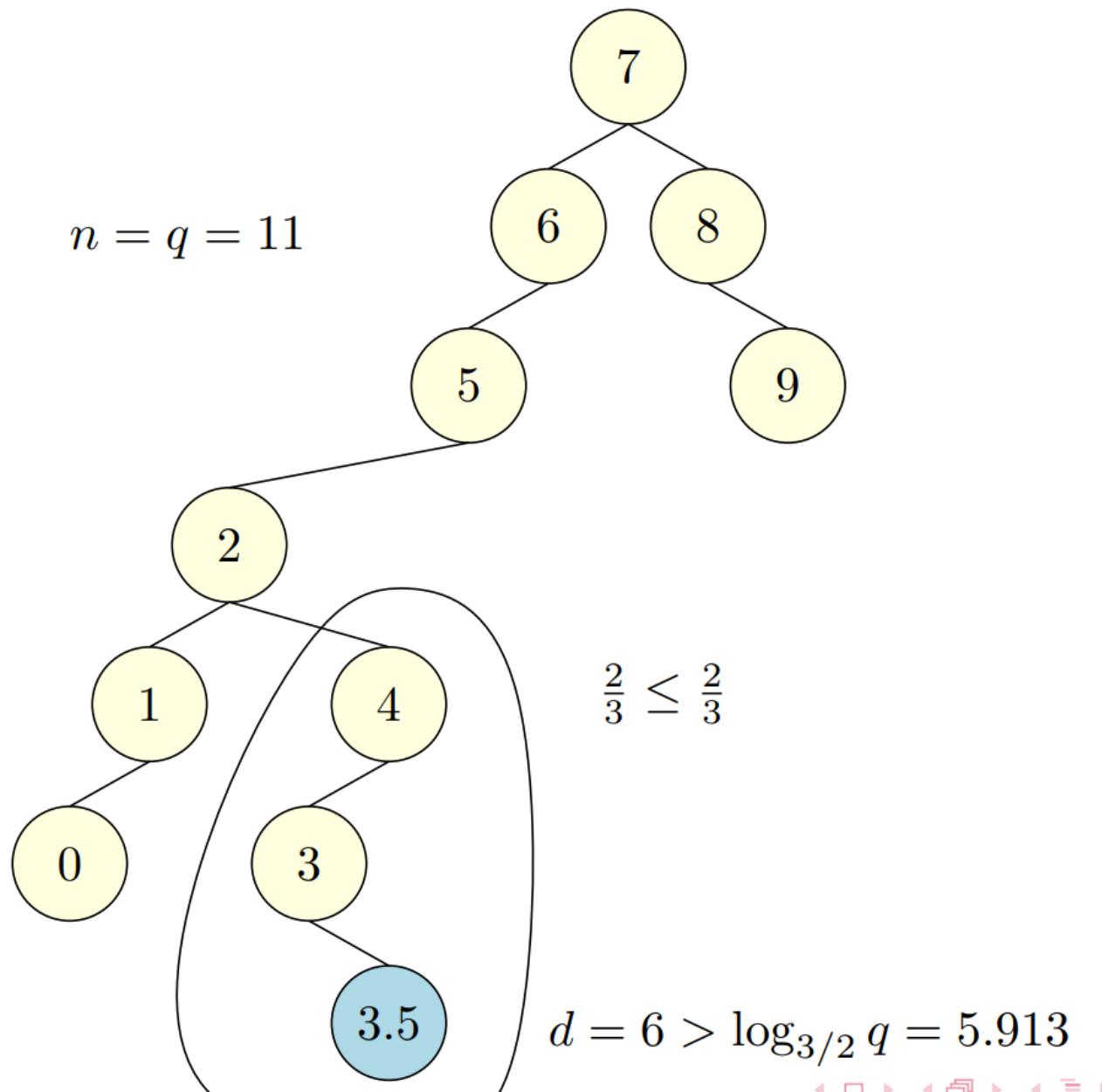


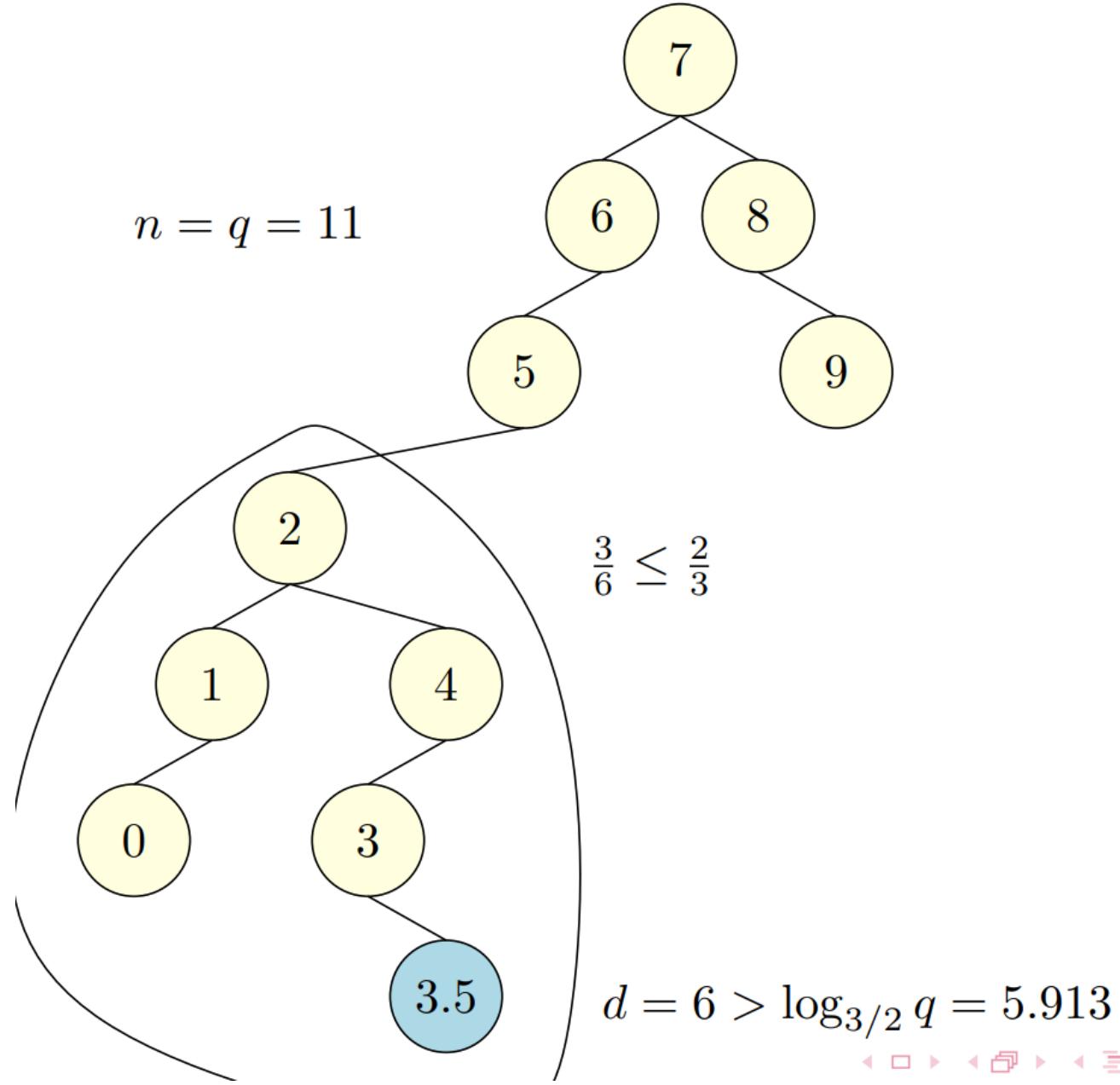




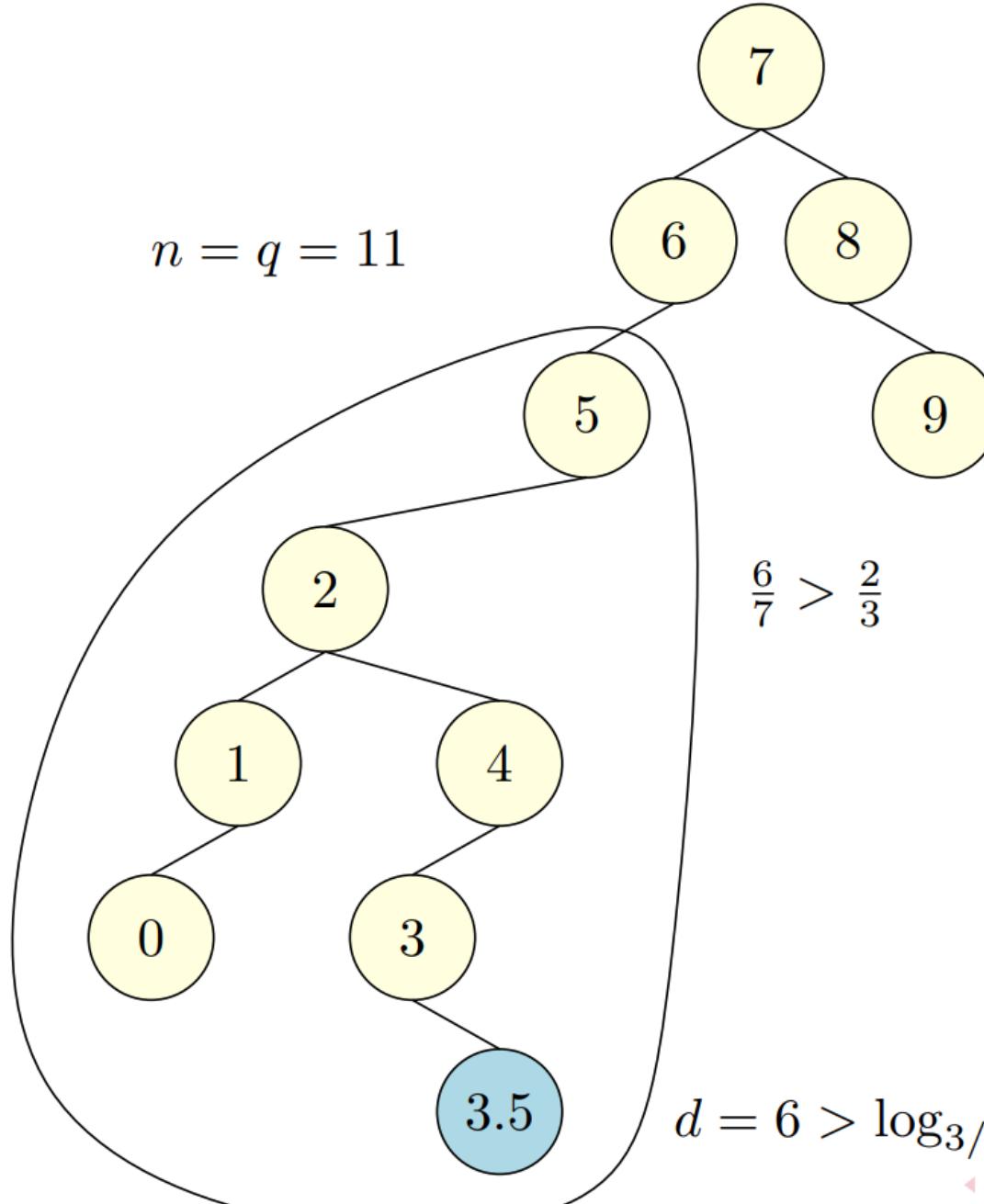


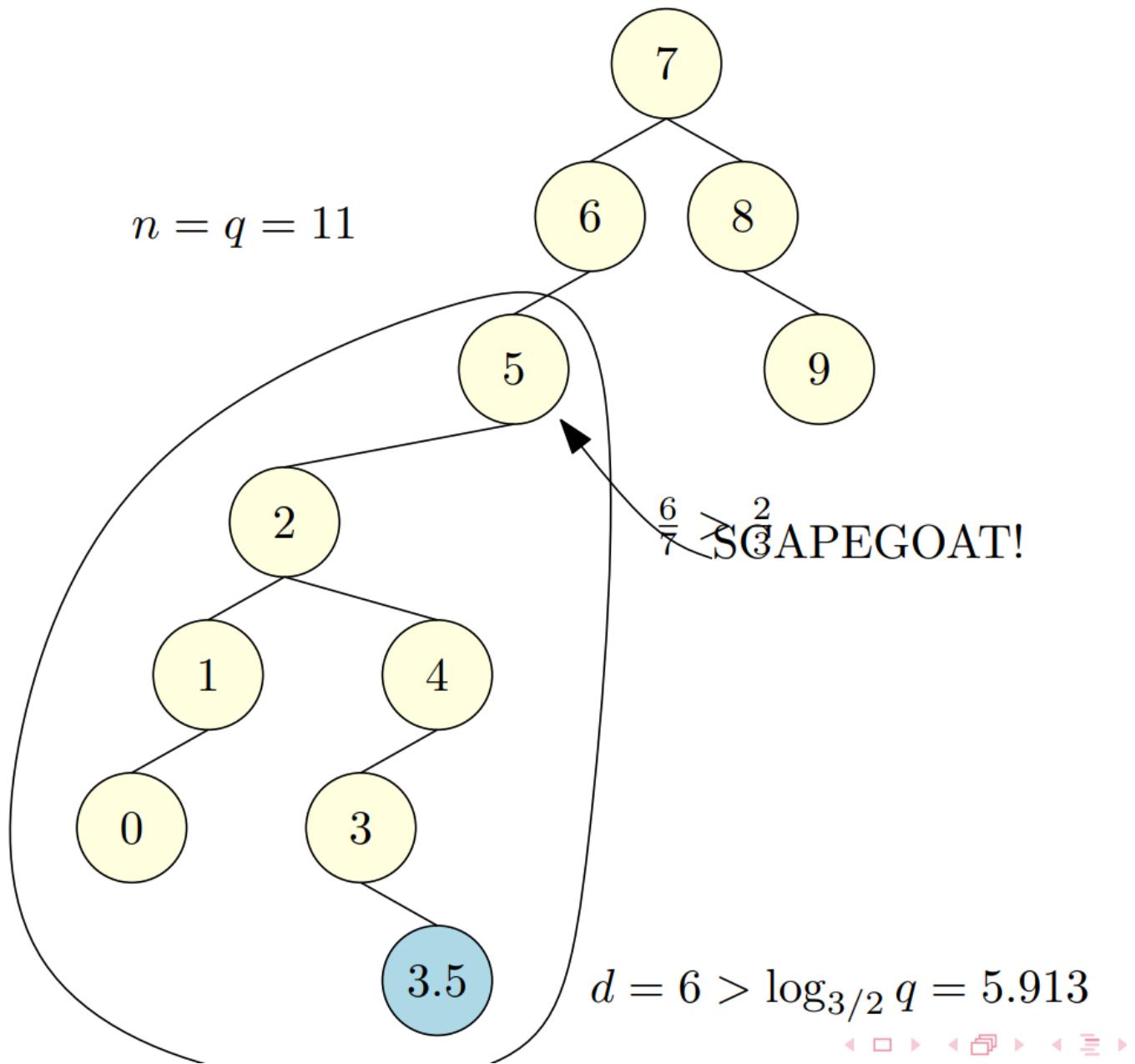


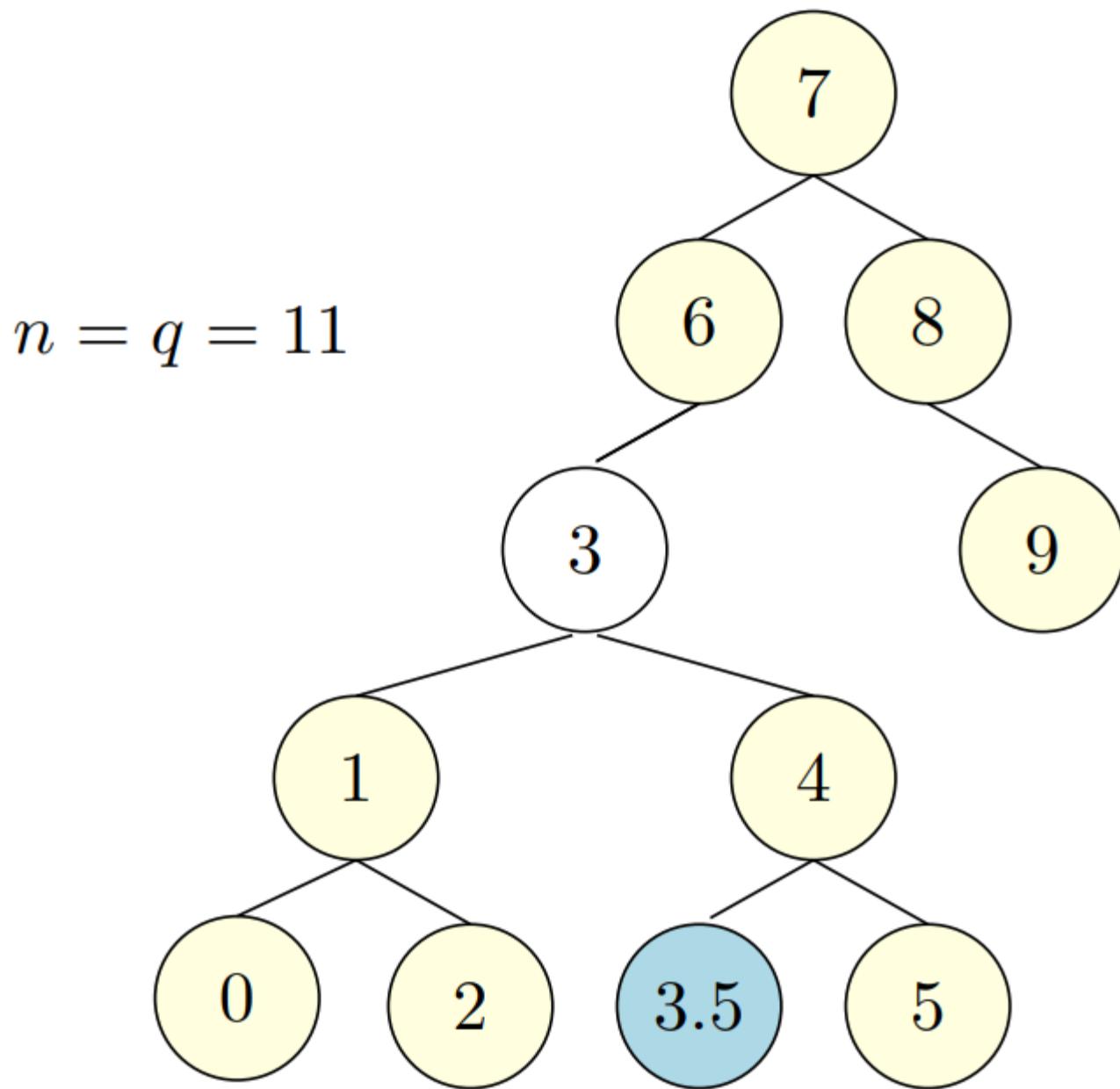




$$n = q = 11$$







```
1. /*
2.  * Java Program to Implement ScapeGoat Tree
3. */
4.
5. import java.util.Scanner;
6.
7. /* Class SGTNode */
8. class SGTNode
9. {
10.     SGTNode right, left, parent;
11.     int value;
12.
13.     /* Constructor */
14.     public SGTNode(int val)
15.     {
16.         value = val;
17.     }
18. }
19.
20. /* Class ScapeGoatTree */
21. class ScapeGoatTree
22. {
23.     private SGTNode root;
24.     private int n, q;
25.
26.     /* Constructor */
27.     public ScapeGoatTree()
28.     {
29.         root = null;
30.         // size = 0
31.         n = 0;
32.     }
33.     /* Function to check if tree is empty */
34.     public boolean isEmpty()
35.     {
36.         return root == null;
37.     }
38.     /* Function to clear tree */
39.     public void makeEmpty()
40.     {
41.         root = null;
42.         n = 0;
43.     }
```

```
44.     /* Function to count number of nodes recursively */
45.     private int size(SGTNode r)
46.     {
47.         if (r == null)
48.             return 0;
49.         else
50.         {
51.             int l = 1;
52.             l += size(r.left);
53.             l += size(r.right);
54.             return l;
55.         }
56.     }
57.     /* Functions to search for an element */
58.     public boolean search(int val)
59.     {
60.         return search(root, val);
61.     }
62.     /* Function to search for an element recursively */
63.     private boolean search(SGTNode r, int val)
64.     {
65.         boolean found = false;
66.         while ((r != null) && !found)
67.         {
68.             int rval = r.value;
69.             if (val < rval)
70.                 r = r.left;
71.             else if (val > rval)
72.                 r = r.right;
73.             else
74.             {
75.                 found = true;
76.                 break;
77.             }
78.             found = search(r, val);
79.         }
80.         return found;
81.     }
```

```
82.     /* Function to return current size of tree */
83.     public int size()
84.     {
85.         return n;
86.     }
87.     /* Function for inorder traversal */
88.     public void inorder()
89.     {
90.         inorder(root);
91.     }
92.     private void inorder(SGTNode r)
93.     {
94.         if (r != null)
95.         {
96.             inorder(r.left);
97.             System.out.print(r.value + " ");
98.             inorder(r.right);
99.         }
100.    }
101.    /* Function for preorder traversal */
102.    public void preorder()
103.    {
104.        preorder(root);
105.    }
106.    private void preorder(SGTNode r)
107.    {
108.        if (r != null)
109.        {
110.            System.out.print(r.value + " ");
111.            preorder(r.left);
112.            preorder(r.right);
113.        }
114.    }
```

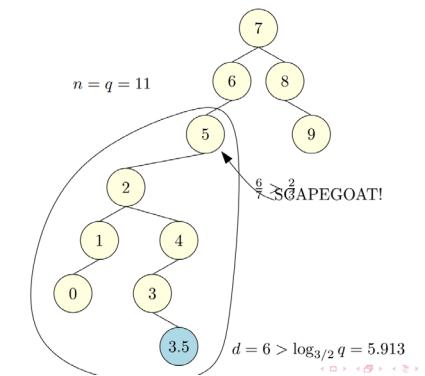
```
115.     /* Function for postorder traversal */
116.     public void postorder()
117.     {
118.         postorder(root);
119.     }
120.     private void postorder(SGTNode r)
121.     {
122.         if (r != null)
123.         {
124.             postorder(r.left);
125.             postorder(r.right);
126.             System.out.print(r.value + " ");
127.         }
128.     }
129.     private static final int log32(int q)
130.     {
131.         final double log23 = 2.4663034623764317;
132.         return (int) Math.ceil(log23 * Math.log(q));
133.     }
134.     /* Function to insert an element */
135.     public boolean add(int x)
136.     {
137.         /* first do basic insertion keeping track of depth */
138.         SGTNode u = new SGTNode(x);
139.         int d = addWithDepth(u);
140.         if (d > log32(q)) {
141.             /* depth exceeded, find scapegoat */
142.             SGTNode w = u.parent;
143.             while (3 * size(w) <= 2 * size(w.parent))
144.                 w = w.parent;
145.             rebuild(w.parent);
146.         }
147.         return d >= 0;
148.     }
```

```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

```

**packIntArray(node\_5, a, 0)**  
**packIntArray(node\_2, a, 0)**  
**packIntArray(node\_1, a, 0)**  
**packIntArray(node\_0, a, 0)**  
**packIntArray(null, a, 0)**



```
149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.     {
152.         int ns = size(u);
153.         SGTNode p = u.parent;
154.         SGTNode[] a = new SGTNode[ns];
155.         packIntoArray(u, a, 0);
156.         if (p == null)
157.         {
158.             root = buildBalanced(a, 0, ns);
159.             root.parent = null;
160.         }
161.         else if (p.right == u)
162.         {
163.             p.right = buildBalanced(a, 0, ns);
164.             p.right.parent = p;
165.         }
166.         else
167.         {
168.             p.left = buildBalanced(a, 0, ns);
169.             p.left.parent = p;
170.         }
171.     }
172.     /* Function to packIntoArray */
173.     protected int packIntoArray(SGTNode u, SGTNode[] a, int i)
174.     {
175.         if (u == null)
176.         {
177.             return i;
178.         }
179.         i = packIntoArray(u.left, a, i);
180.         a[i++] = u;
181.         return packIntoArray(u.right, a, i);
182.     }
```

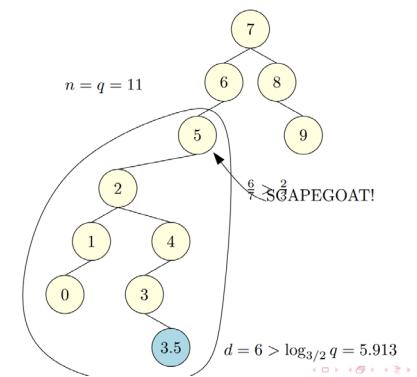
`packIntoArray(node_5, a, 0)`

**packIntoArray(node\_2, a, 0)**

`packIntoArray(node_1, a, 0)`

`packIntoArray(node_0, a, 0)`

packIntoArray(null, a, 0) Return 0



```
149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.     {
152.         int ns = size(u);
153.         SGTNode p = u.parent;
154.         SGTNode[] a = new SGTNode[ns];
155.         packIntoArray(u, a, 0);
156.         if (p == null)
157.         {
158.             root = buildBalanced(a, 0, ns);
159.             root.parent = null;
160.         }
161.         else if (p.right == u)
162.         {
163.             p.right = buildBalanced(a, 0, ns);
164.             p.right.parent = p;
165.         }
166.         else
167.         {
168.             p.left = buildBalanced(a, 0, ns);
169.             p.left.parent = p;
170.         }
171.     }
172.     /* Function to packIntoArray */
173.     protected int packIntoArray(SGTNode u, SGTNode[] a, int i)
174.     {
175.         if (u == null)
176.         {
177.             return i;
178.         }
179.         i = packIntoArray(u.left, a, i);
180.         a[i++] = u;
181.         return packIntoArray(u.right, a, i);
182.     }
```

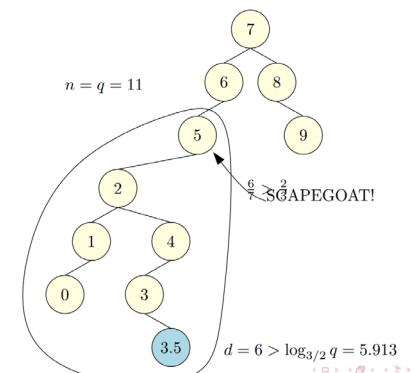
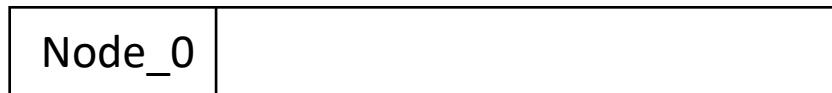
`packIntoArray(node_5, a, 0)`

**packIntoArray(node\_2, a, 0)**

`packIntoArray(node_1, a, 0)`

`packIntoArray(node_0, a, 0)`

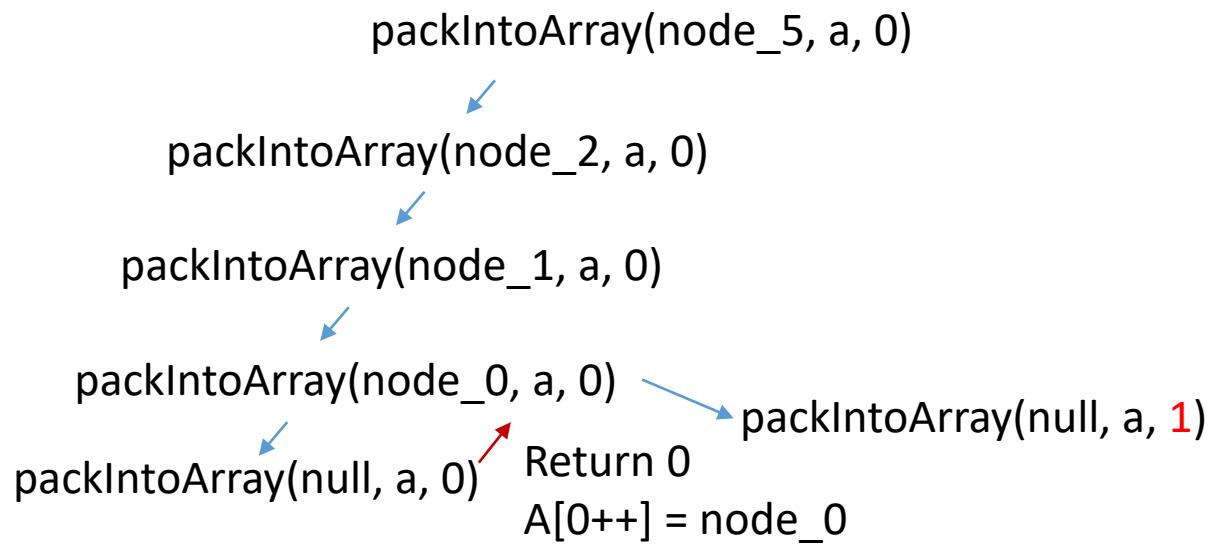
packIntoArray(null, a, 0) Return 0  
A[0++] = node\_0



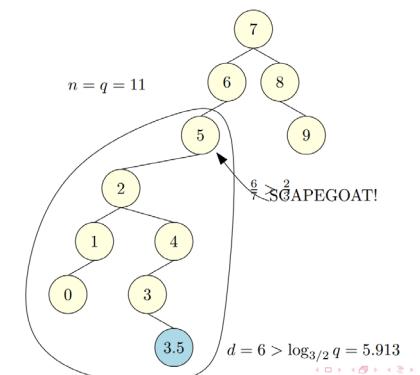
```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

```



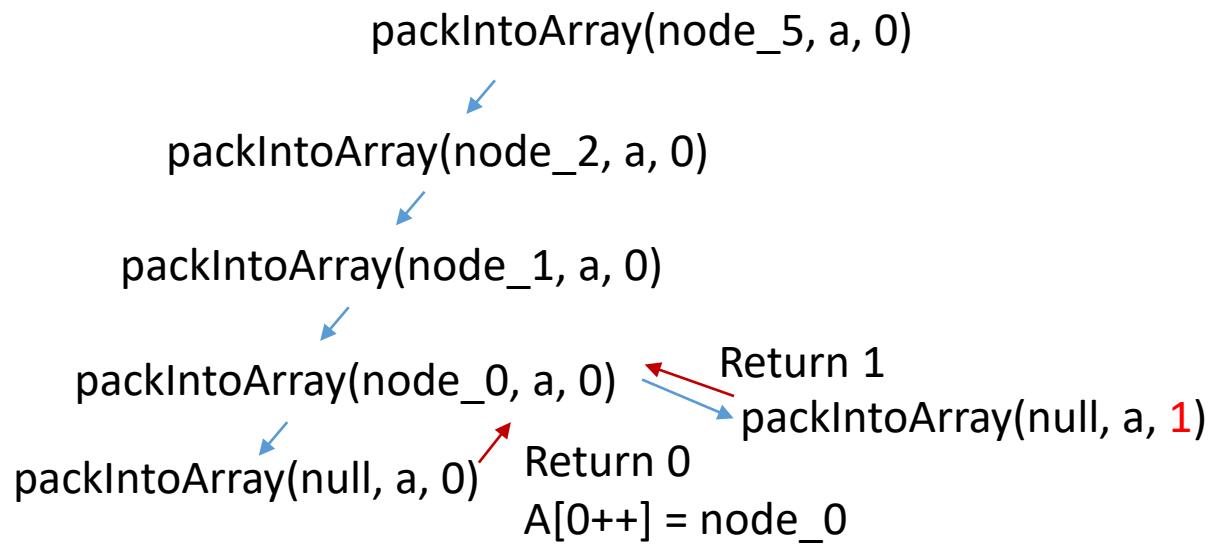
Node\_0



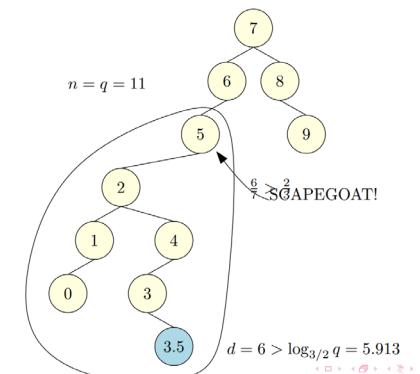
```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

```



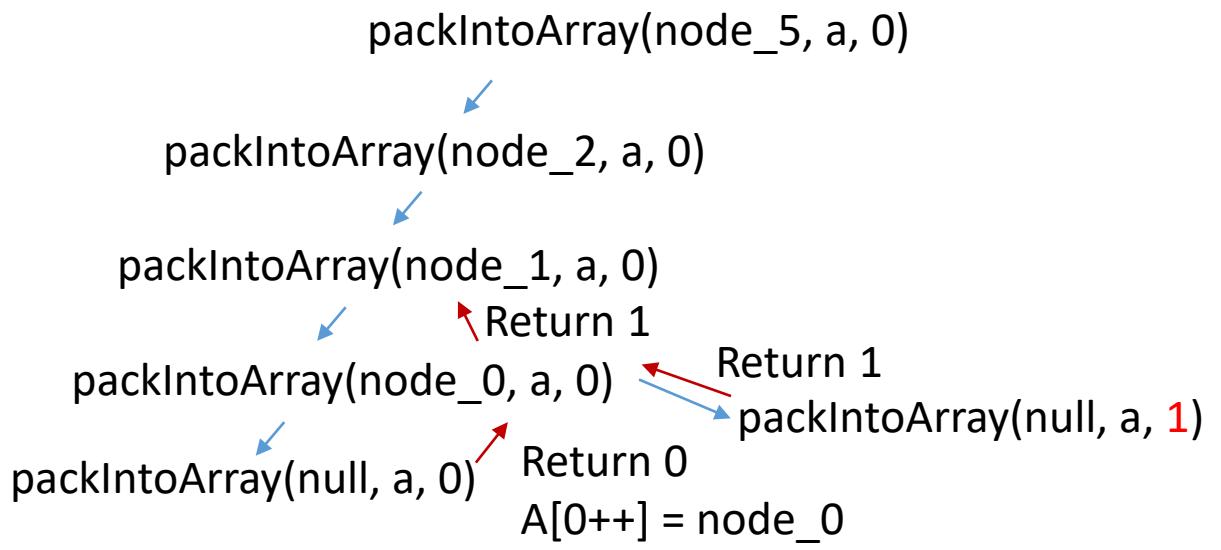
Node\_0



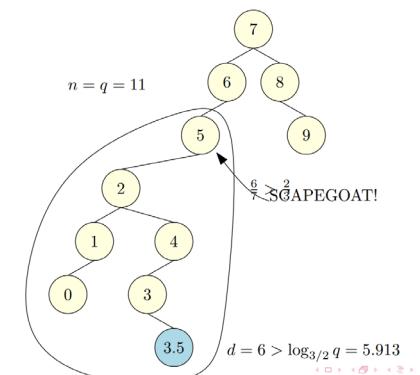
```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

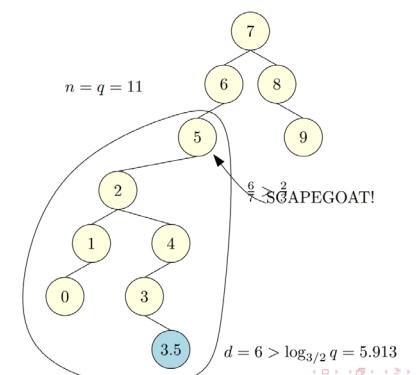
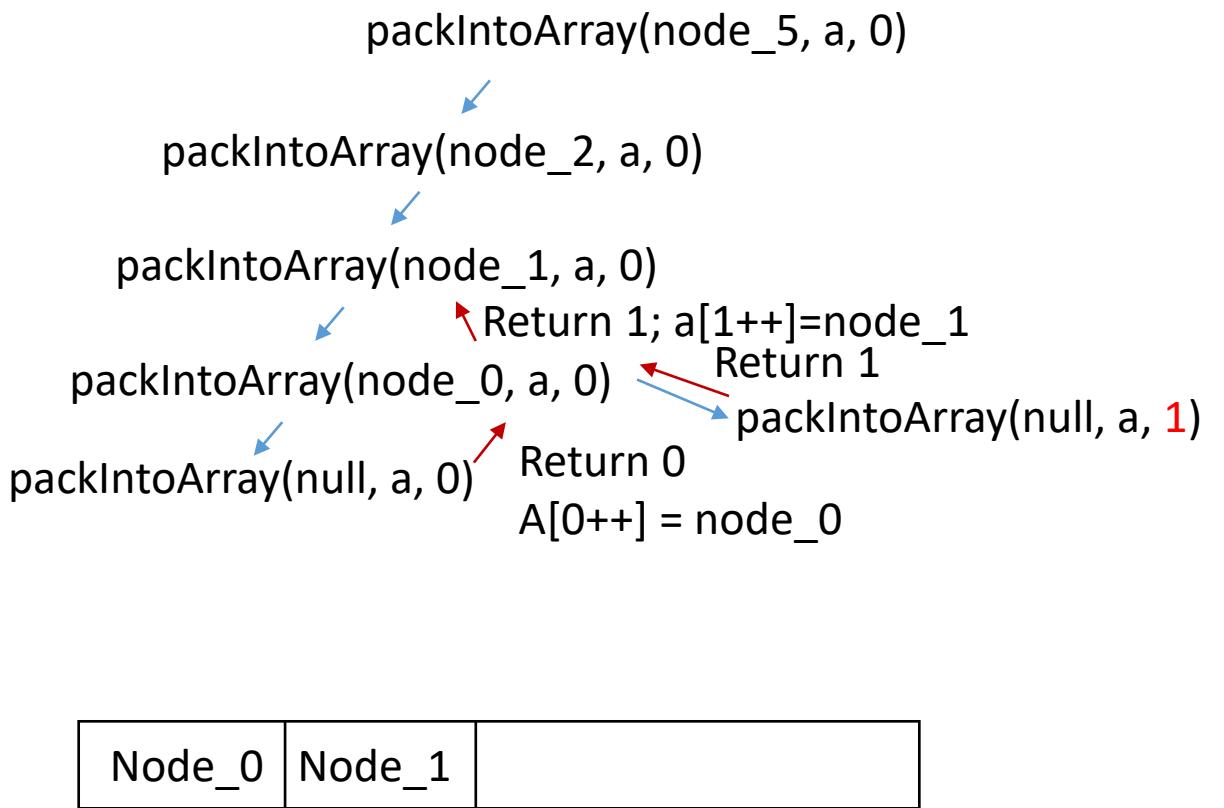
```



Node\_0



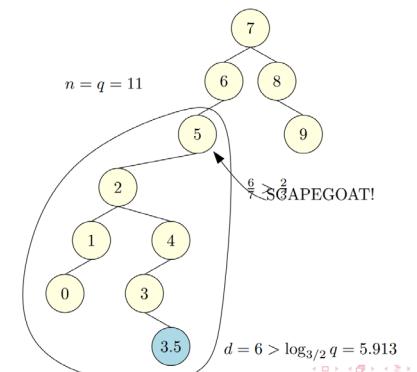
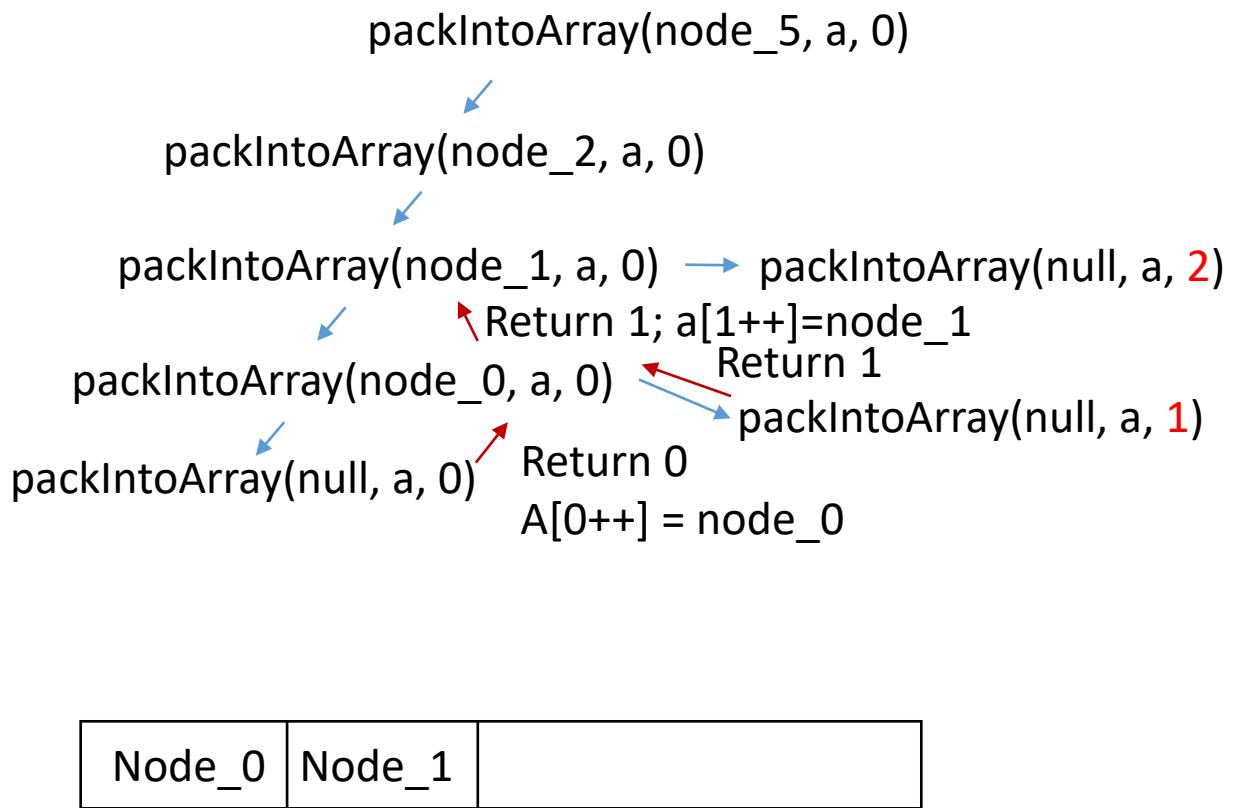
```
149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.     {
152.         int ns = size(u);
153.         SGTNode p = u.parent;
154.         SGTNode[] a = new SGTNode[ns];
155.         packIntoArray(u, a, 0);
156.         if (p == null)
157.         {
158.             root = buildBalanced(a, 0, ns);
159.             root.parent = null;
160.         }
161.         else if (p.right == u)
162.         {
163.             p.right = buildBalanced(a, 0, ns);
164.             p.right.parent = p;
165.         }
166.         else
167.         {
168.             p.left = buildBalanced(a, 0, ns);
169.             p.left.parent = p;
170.         }
171.     }
172.     /* Function to packIntoArray */
173.     protected int packIntoArray(SGTNode u, SGTNode[] a, int i)
174.     {
175.         if (u == null)
176.         {
177.             return i;
178.         }
179.         i = packIntoArray(u.left, a, i);
180.         a[i++] = u;
181.         return packIntoArray(u.right, a, i);
182.     }
```



```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

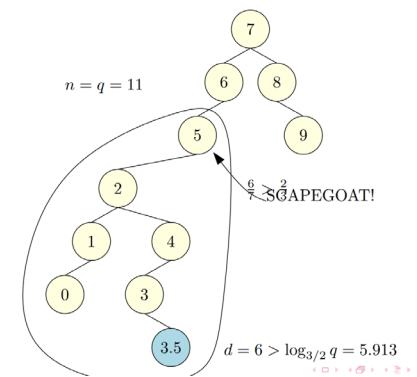
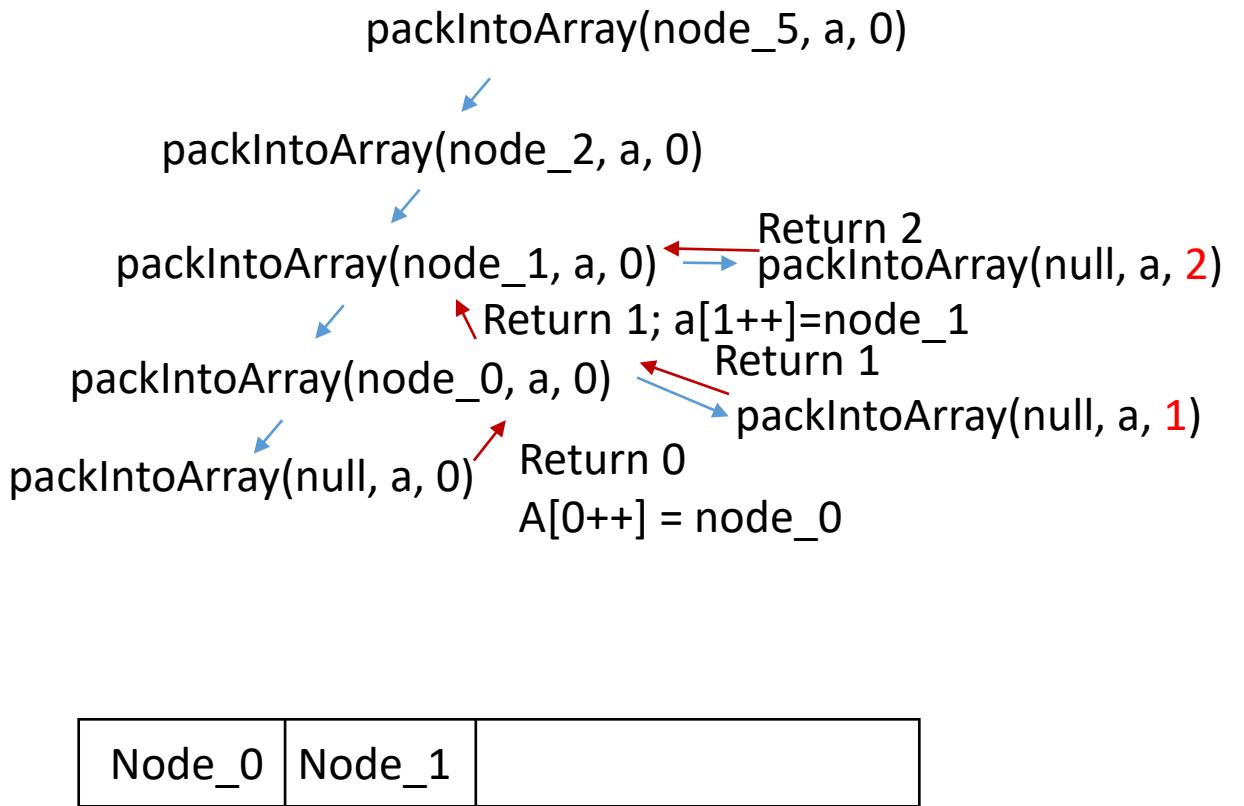
```



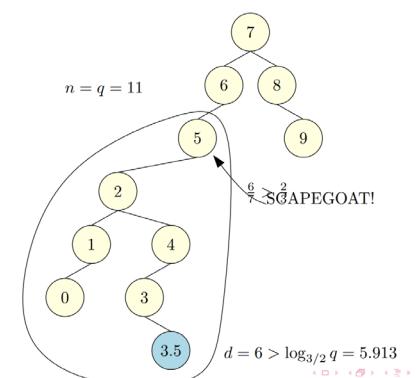
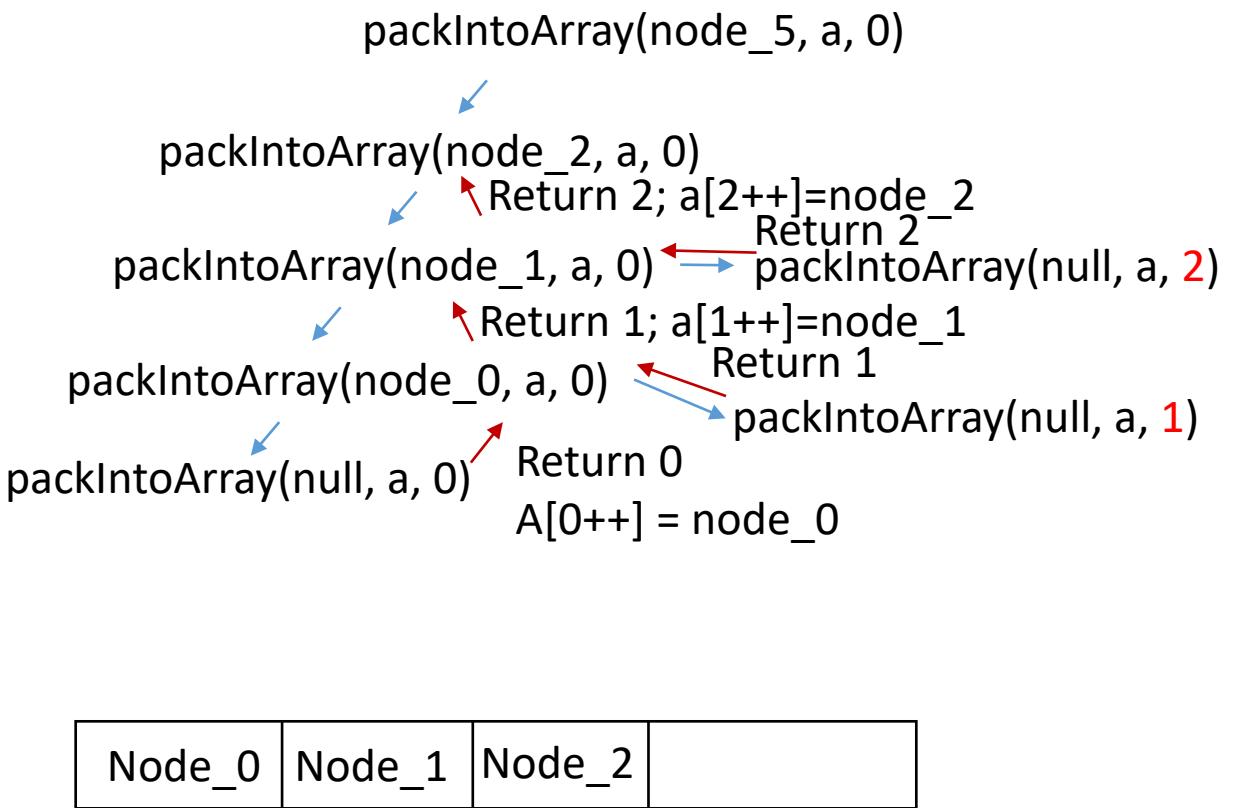
```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.    {
152.        int ns = size(u);
153.        SGTNode p = u.parent;
154.        SGTNode[] a = new SGTNode[ns];
155.        packIntArray(u, a, 0);
156.        if (p == null)
157.        {
158.            root = buildBalanced(a, 0, ns);
159.            root.parent = null;
160.        }
161.        else if (p.right == u)
162.        {
163.            p.right = buildBalanced(a, 0, ns);
164.            p.right.parent = p;
165.        }
166.        else
167.        {
168.            p.left = buildBalanced(a, 0, ns);
169.            p.left.parent = p;
170.        }
171.    }
172.    /* Function to packIntArray */
173.    protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.    {
175.        if (u == null)
176.        {
177.            return i;
178.        }
179.        i = packIntArray(u.left, a, i);
180.        a[i++] = u;
181.        return packIntArray(u.right, a, i);
182.    }

```



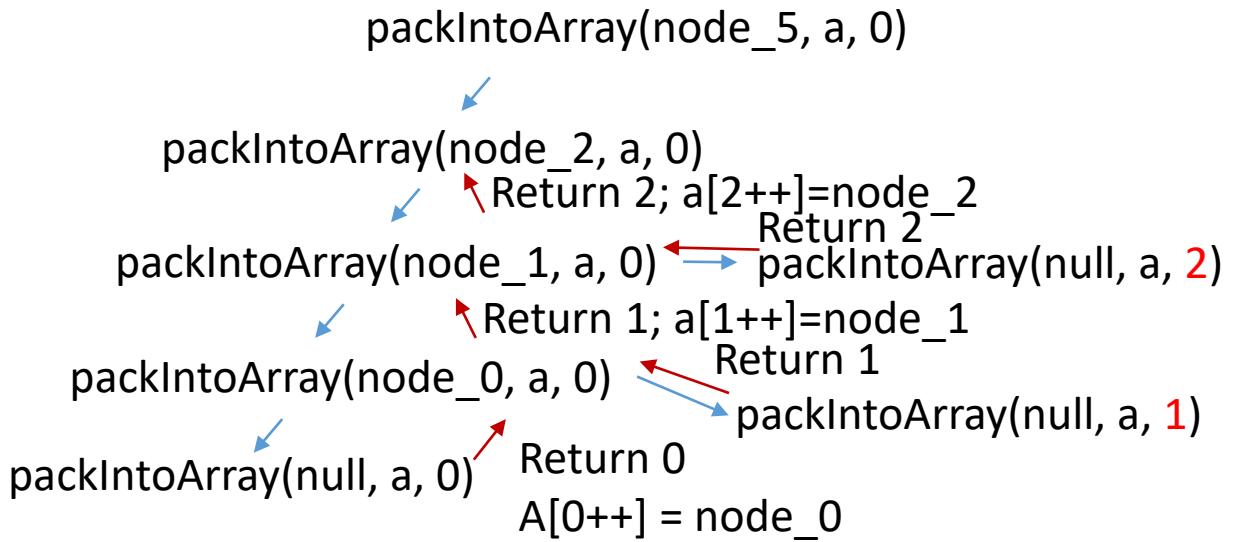
```
149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.     {
152.         int ns = size(u);
153.         SGTNode p = u.parent;
154.         SGTNode[] a = new SGTNode[ns];
155.         packIntoArray(u, a, 0);
156.         if (p == null)
157.         {
158.             root = buildBalanced(a, 0, ns);
159.             root.parent = null;
160.         }
161.         else if (p.right == u)
162.         {
163.             p.right = buildBalanced(a, 0, ns);
164.             p.right.parent = p;
165.         }
166.         else
167.         {
168.             p.left = buildBalanced(a, 0, ns);
169.             p.left.parent = p;
170.         }
171.     }
172.     /* Function to packIntoArray */
173.     protected int packIntoArray(SGTNode u, SGTNode[] a, int i)
174.     {
175.         if (u == null)
176.         {
177.             return i;
178.         }
179.         i = packIntoArray(u.left, a, i);
180.         a[i++] = u;
181.         return packIntoArray(u.right, a, i);
182.     }
```



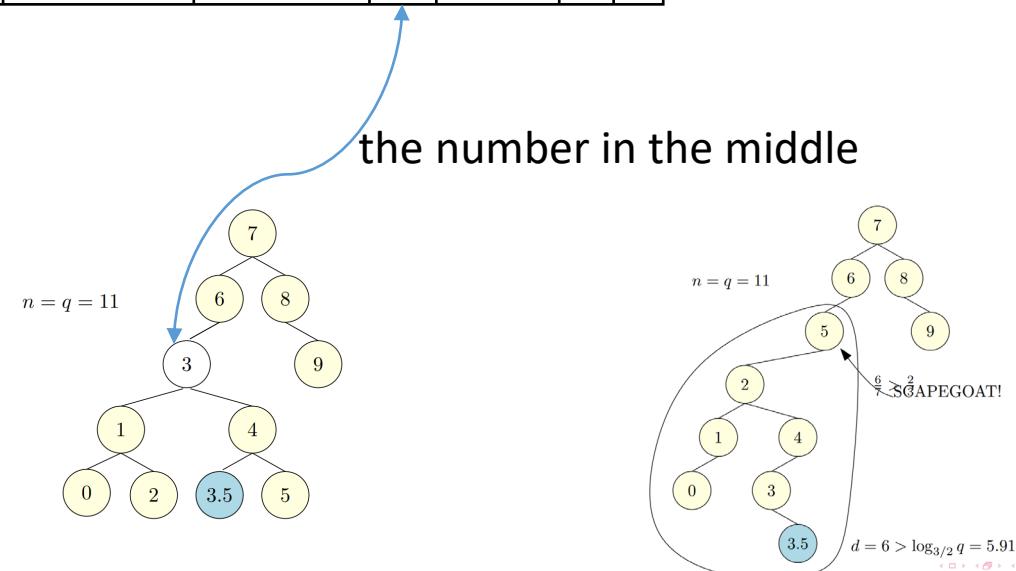
```

149.     /* Function to rebuild tree from node u */
150.     protected void rebuild(SGTNode u)
151.     {
152.         int ns = size(u);
153.         SGTNode p = u.parent;
154.         SGTNode[] a = new SGTNode[ns];
155.         packIntArray(u, a, 0);
156.         if (p == null)
157.         {
158.             root = buildBalanced(a, 0, ns);
159.             root.parent = null;
160.         }
161.         else if (p.right == u)
162.         {
163.             p.right = buildBalanced(a, 0, ns);
164.             p.right.parent = p;
165.         }
166.         else
167.         {
168.             p.left = buildBalanced(a, 0, ns);
169.             p.left.parent = p;
170.         }
171.     }
172.     /* Function to packIntArray */
173.     protected int packIntArray(SGTNode u, SGTNode[] a, int i)
174.     {
175.         if (u == null)
176.         {
177.             return i;
178.         }
179.         i = packIntArray(u.left, a, i);
180.         a[i++] = u;
181.         return packIntArray(u.right, a, i);
182.     }

```



|        |        |        |   |     |   |   |
|--------|--------|--------|---|-----|---|---|
| Node_0 | Node_1 | Node_2 | 3 | 3.5 | 4 | 5 |
|--------|--------|--------|---|-----|---|---|



```
183.     /* Function to build balanced nodes */
184.     protected SGTNode buildBalanced(SGTNode[] a, int i, int ns)
185.     {
186.         if (ns == 0)
187.             return null;
188.         int m = ns / 2;
189.         a[i + m].left = buildBalanced(a, i, m);
190.         if (a[i + m].left != null)
191.             a[i + m].left.parent = a[i + m];
192.         a[i + m].right = buildBalanced(a, i + m + 1, ns - m - 1);
193.         if (a[i + m].right != null)
194.             a[i + m].right.parent = a[i + m];
195.         return a[i + m];
196.     }
```

```
197.  /* Function add with depth */
198.  public int addWithDepth(SGTNode u)
199.  {
200.      SGTNode w = root;
201.      if (w == null)
202.      {
203.          root = u;
204.          n++;
205.          q++;
206.          return 0;
207.      }
208.      boolean done = false;
209.      int d = 0;
210.      do {
211.
212.          if (u.value < w.value)
213.          {
214.              if (w.left == null)
215.              {
216.                  w.left = u;
217.                  u.parent = w;
218.                  done = true;
219.              }
220.              else
221.              {
222.                  w = w.left;
223.              }
224.          }
225.          else if (u.value > w.value)
226.          {
227.              if (w.right == null)
228.              {
229.                  w.right = u;
230.                  u.parent = w;
231.                  done = true;
232.              }
233.              w = w.right;
234.          }
235.          else
236.          {
237.              return -1;
238.          }
239.          d++;
240.      } while (!done);
241.      n++;
242.      q++;
243.      return d;
244.  }
245. }
```

```
247. public class ScapeGoatTreeTest
248. {
249.     public static void main(String[] args)
250.     {
251.         Scanner scan = new Scanner(System.in);
252.         /* Creating object of ScapeGoatTree */
253.         ScapeGoatTree sgt = new ScapeGoatTree();
254.         System.out.println("ScapeGoat Tree Test\n");
255.         char ch;
256.         /* Perform tree operations */
257.         do
258.         {
259.             System.out.println("\nScapeGoat Tree Operations\n");
260.             System.out.println("1. insert ");
261.             System.out.println("2. count nodes");
262.             System.out.println("3. search");
263.             System.out.println("4. check empty");
264.             System.out.println("5. make empty");
265.
266.             int choice = scan.nextInt();
267.             switch (choice)
268.             {
269.                 case 1 :
270.                     System.out.println("Enter integer element to insert");
271.                     sgt.add( scan.nextInt() );
272.                     break;
273.                 case 2 :
274.                     System.out.println("Nodes = "+ sgt.size());
275.                     break;
276.                 case 3 :
277.                     System.out.println("Enter integer element to search");
278.                     System.out.println("Search result : "+ sgt.search( scan.nextInt() ) )
279.                     break;
280.                 case 4 :
281.                     System.out.println("Empty status = "+ sgt.isEmpty());
282.                     break;
283.                 case 5 :
284.                     System.out.println("\nTree cleared\n");
285.                     sgt.makeEmpty();
286.                     break;
287.                 default :
288.                     System.out.println("Wrong Entry \n ");
289.                     break;
290.             }
291.         }
292.     }
293. }
```

```
291.         /* Display tree */
292.         System.out.print("\nPost order : ");
293.         sgt.postorder();
294.         System.out.print("\nPre order : ");
295.         sgt.preorder();
296.         System.out.print("\nIn order : ");
297.         sgt.inorder();
298.
299.         System.out.println("\nDo you want to continue (Type y or n) \n");
300.         ch = scan.next().charAt(0);
301.     } while (ch == 'Y' || ch == 'y');
302. }
303. }
```

# Hashmap class in Java

- Provide implementation of the Map interface
- Stores data in (Key, Value) pairs
- To access a value, you must know its key
- It does not allow duplicate keys but allows duplicate values
- It allows null key also but only once and multiple null values
- It makes no guarantees as to the order of the map
- It is similar to HashTable but is unsynchronized

```
1 // Java program to illustrate
2 // Java.util.HashMap
3
4 import java.util.HashMap;
5 import java.util.Map;
6
7 public class GFG {
8     public static void main(String[] args)
9     {
10
11         HashMap<String, Integer> map
12             = new HashMap<>();
13
14         print(map);
15         map.put("vishal", 10);
16         map.put("sachin", 30);
17         map.put("vaibhav", 20);
18
19         System.out.println("Size of map is:- "
20                         + map.size());
21
22         print(map);
23         if (map.containsKey("vishal")) {
24             Integer a = map.get("vishal");
25             System.out.println("value for key"
26                             + " \"vishal\" is:- "
27                             + a);
28         }
29
30         map.clear();
31         print(map);
32     }
33
34     public static void print(Map<String, Integer> map)
35     {
36         if (map.isEmpty()) {
37             System.out.println("map is empty");
38         }
39
40         else {
41             System.out.println(map);
42         }
43     }
44 }
45 }
```

# Hashtable class in Java

- It is similar to HashMap, but it is synchronized
- It stores key/value pairs

```

1 // Java code to illustrate the contains() method
2 import java.util.*;
3
4 public class Hash_Table_Demo {
5     public static void main(String[] args)
6     {
7
8         // Creating an empty Hashtable
9         Hashtable<Integer, String> hash_table =
10            new Hashtable<Integer, String>();
11
12        // Mapping string values to int keys
13        hash_table.put(10, "Geeks");
14        hash_table.put(15, "4");
15        hash_table.put(20, "Geeks");
16        hash_table.put(25, "Welcomes");
17        hash_table.put(30, "You");
18
19        // Displaying the HashMap
20        System.out.println("Initial Table is: " + hash_table);
21
22        // Checking for the Value 'Geeks'
23        System.out.println("Is the value 'Geeks' present? " +
24                           hash_table.contains("Geeks"));
25
26        // Checking for the Value 'World'
27        System.out.println("Is the value 'World' present? " +
28                           hash_table.contains("World"));
29    }
30
31

```

**contains(Object value):** The `java.util.Hashtable.contains(Object value)` method in Java is used to check whether a particular value is being mapped by any keys present in the Hashtable.

**Syntax:** `Hash_table.contains(Object value)`

**Parameters:** The method accepts one parameter `value` of object type and refers to the value of the hashtable whose mapping is to be verified.

**Return Value:** The method returns a `boolean` true value if the passed value is mapped by any of the keys in the Hashtable.

**Exceptions:** The method throws `NullPointerException` if the passed value is Null.

**Output:**

```

Initial Table is: {10=Geeks, 20=Geeks, 30=You, 15=4, 25=Welcomes}
Is the value 'Geeks' present? true
Is the value 'World' present? false

```

```
1 // Java code illustrating elements() method
2 import java.util.*;
3 class hashTabledemo {
4     public static void main(String[] arg)
5     {
6         // creating a hash table
7         Hashtable<Integer, String> h =
8             new Hashtable<Integer, String>();
9
10        h.put(3, "Geeks");
11        h.put(2, "forGeeks");
12        h.put(1, "isBest");
13
14        // create enumeration
15        Enumeration e = h.elements();
16
17        System.out.println("display values:");
18
19        while (e.hasMoreElements()) {
20            System.out.println(e.nextElement());
21        }
22    }
23}
24}
```

**Enumeration elements()** :Returns an enumeration of the values obtained in hash table.

**Syntax** :public Enumeration elements()

**Returns** :returns an enumeration of the values in this hash table.

**Exception** :NA<

Output:

```
display values:
Geeks
forGeeks
isBest
```

```
1 // Java code illustrating get() method
2 import java.util.*;
3 class Vector_demo {
4     public static void main(String[] arg)
5     {
6
7         // creating a hash table
8         Hashtable<String, Integer> marks =
9             new Hashtable<String, Integer>();
10
11        // enter name/marks pair
12        marks.put("tweener", new Integer(345));
13        marks.put("krantz", new Integer(245));
14        marks.put("burrows", new Integer(790));
15        marks.put("tancredi", new Integer(365));
16        marks.put("bellick", new Integer(435));
17
18        // get the value mapped with key krantz
19
20        System.out.println(marks.get("krantz"));
21    }
22 }
23 }
```

**Object get(Object key)** : used to get the object that contains the value associated with key.

**Syntax** :public Object get(Object key)

**Returns** :the value to which the key is mapped in this hashtable.

**Exception** :NullPointerException if the key is null.

Output:

**boolean isEmpty()** :used to test if this hashtable maps no keys to values.

**Syntax** :public boolean isEmpty()

**Returns** :true if this hashtable maps no keys to values; false otherwise.

**Exception** :NA

**Enumeration keys()** :used to get enumeration of the keys contained in the hash table.

**Syntax** :public Enumeration keys()

**Returns** :an enumeration of the keys in this hashtable.

**Exception** :NA

**KeySet()** :used to get a Set view of the keys contained in this hash table.

**Syntax** :public Set keySet()

**Returns** :a set view of the keys contained in this map.

**Exception** :NA

**void putAll(Map t)** :copies all of the mappings from the specified map to this hashtable.

**Syntax** :public void putAll(Map t)

**Returns** :NA

**Exception** :NullPointerException if the specified map is null.

**Object remove(Object key)** :Removes key and its value.

**Syntax** :public Object remove(Object key)

**Returns** :returns the value associated with key. If key is not in the hash table, a null object is returned.

**Exception** :NullPointerException if specified key is null.

**int size()** :returns the number of entries in hash table.

**Syntax** :public int size()

**Returns** :returns the number of keys in this hashtable.

**Exception** :NA

**String toString()** :returns the string equivalent of a hash table.

**Syntax** :public String toString()

**Returns** :

**Exception** :NA

**values()** :used to get a Collection view of the values contained in this Hashtable.

**Syntax** :public Collection values()

**Returns** :returns a collection view of the values contained in this map.

**Exception** :NA

# Acknowledgement

- Some slides are from Pat Morin from Carleton University