



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

# Important Notice

- The lecture slides are provided so that you can focus on understanding the material as it is being discussed and **take supplementary notes**.
- Additional material **will** be covered in lectures that does not appear in these notes.
- Getting the notes is **not** a substitute for attending lectures!
- Sample programs will be available for download on Blackboard.
- These notes were originally developed by Prof. Michael Madden and they have been updated/extended by Dr. Frank Glavin.



# CT2109 Learning Objectives

1. Explain the structure, properties and use of data structures including Linked Lists, Stacks, Queues, Trees, Binary Search Trees, and Graphs, including algorithms to process them.
2. Implement these data structures, use the implementations, and explain them.
3. Explain and apply the concepts of recursion and dynamic programming.
4. Define, discuss, and apply the concept of O-notation as related to algorithmic complexity. Contrast this to other notations such as big Omega & big Theta,
5. Define and provide examples for P, NP, NP-Hard and NP-Complete problems.
6. Explain the general operation and algorithm details of a variety of sorting algorithms and implement and analyse them.
7. Explain the concepts of lossy and lossless compression and describe and implement lossless compression algorithms.
8. Analyse the space and time requirements of any algorithm encountered in this module, theoretically and empirically.
9. Evaluate algorithms and data structures, analyse their complexity, discuss their relative merits, and make rational choices about which is best for an application.
10. Create your own algorithmic solutions to problems and implement them in Java.



# CT2109 Module Content 1/2

## List-Based Abstract Data Structures

Queues; Stacks; Linked lists

Design, OOP implementation, analysis

## Algorithm Analysis

Speed and storage

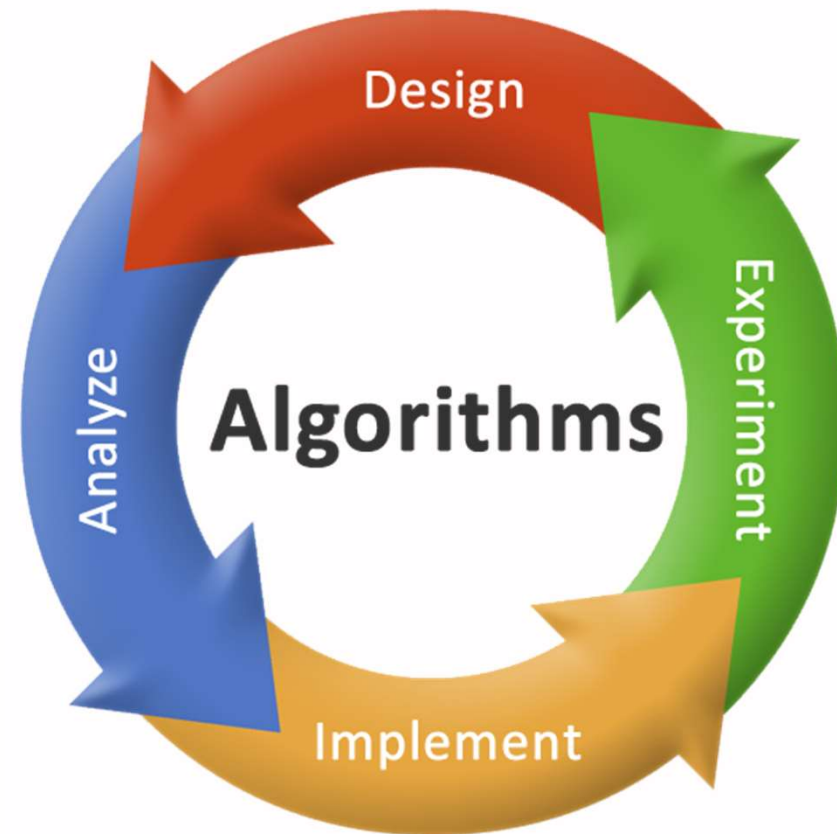
Dynamic Programming

Theoretical & empirical analysis

## Sorting Algorithms

Quick Sort, Shell Sort, Radix Sort, others

Design, OOP implementation, analysis



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY



# CT2109 Module Content 2/2

## Tree Structures

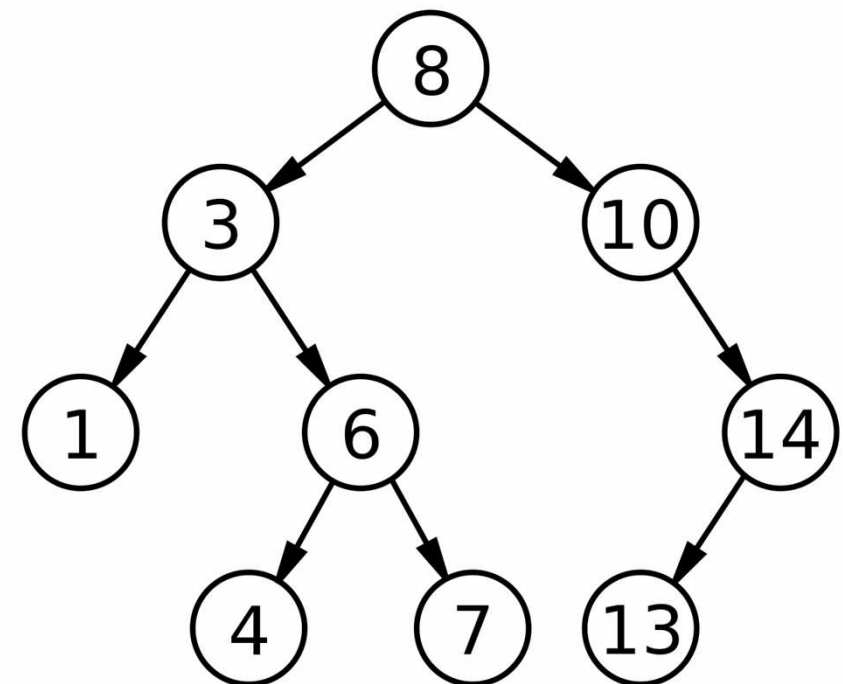
Binary trees; tree algorithms; depth-first and breadth-first searching; balanced trees; AVL trees  
Design, OOP implementation, analysis

## Data Compression

Huffman Encoding

## Practical Considerations [cross-cutting theme]

Analysing, choosing, implementing & applying algorithms & data structures in useful programs.



# We Assume You Studied CT2106 ...

## Java Language

Syntax, Important libraries, Strings, Arrays, ArrayLists  
Coding conventions, Numeric precision

## Object Oriented Programming

Composition, Inheritance, Polymorphism, Abstraction,  
Constructors, Casting, Interfaces, Abstract Classes,  
Java Collections Framework

## Good Practice

Debugging, Exception handling, Unit testing,  
Test-driven development, Design Patterns



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

## Teaching and Assessment

### Assessment:

Lab assignments (30% weight)

**Written exam in Summer (70%)**

### Lectures:

Lecture 1: Friday 9:00 – 9:50

Lecture 2: Friday, 10:00 – 10:50

### Lab sessions:

2 hours/week starting in Week 2:

**Thursdays** 12:00-14:00 (2BLE, 2BP, others)

**Fridays** 12:00-14:00 (2BCT)



# Resources

## Suggested Reading:

"*Data Structures and Algorithms In Java*",  
Goodrich & Tamassia

"*Data Structures and Abstractions with Java*",  
Carrano & Savitch

"*Java: How to Program*", Deitel & Deitel  
(General-purpose Java book)

Others: see library

Notes **and** lectures!

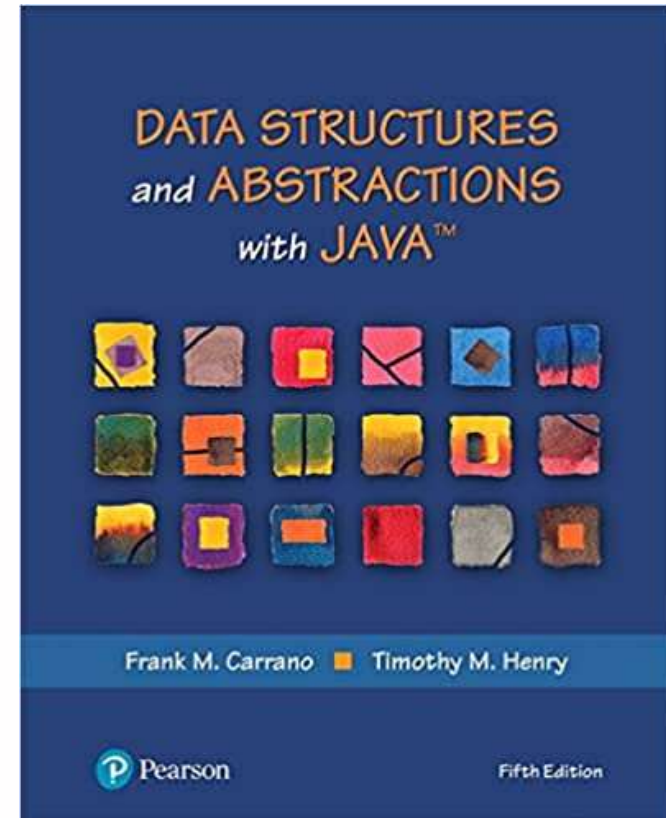
## Course Website:

<http://nuigalway.blackboard.com>

Sample programs, assignments, announcements

## Java IDEs:

<https://hackr.io/blog/best-java-ides>



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY






OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# CT2109

## Topic 1: Stacks and Queues

University  
ofGalway.ie

- 
- Define what an Abstract Data Type is
  - Explain the purpose and use of Stack and Queue Abstract Data Types

- Implement Stack and Queue based on Arrays, including basic and more advanced implementation details
- Demonstrate an ability and understanding of using Stacks and Queues in applications



## What You'll Achieve in this Topic

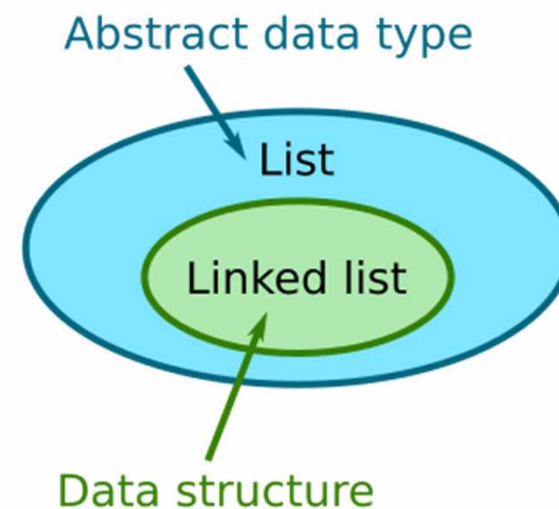
# Abstract Data Type (ADT) 1/2

An abstract model of a data structure, that specifies

- Data stored
- Operations that may be performed on data

## Composite ADTs

- Used to manage *collections* of data
- Array, List, Stack, Queue, Hash Table, ...





# Abstract Data Type (ADT) 2/2

ADT Specifies *what* each operation does, but not *how*

In OOP languages like Java, corresponds naturally to an **Interface Definition**

An ADT is *realised* as a concrete data structure

In Java, this is a class that **implements** the interface

ADT specification of a list?



# Stacks & Queues: Overview

Stacks and Queues:  
Linearly ordered ADTs for list-structured data

## Stack:

### Last In, First Out:

Items can only enter/leave via the *top*

**Push** and **Pop** to add and remove

Example Applications:

Processing nested structures

'Undo' operation in editor



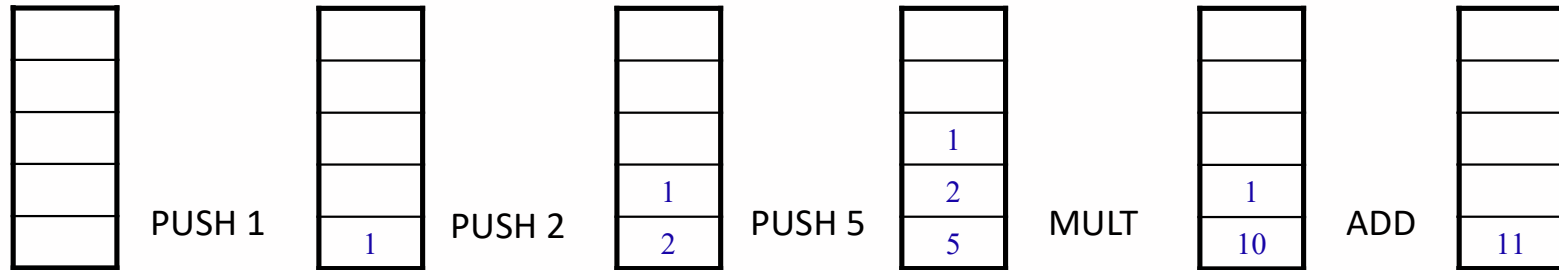
OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Stack Example: A Stack-Machine

Assignment / simple expression written in a high-level programming language (e.g. C):  
 $a = 1 + (2 * 5);$

Machine instructions after compilation:

PUSH 1  
PUSH 2  
PUSH 5  
MULT  
ADD  
...



# Stacks & Queues: Overview

## Queue:

### First In, First Out:

Items enter at the rear, leave at the front

**Enqueue** and **Dequeue** to add and remove

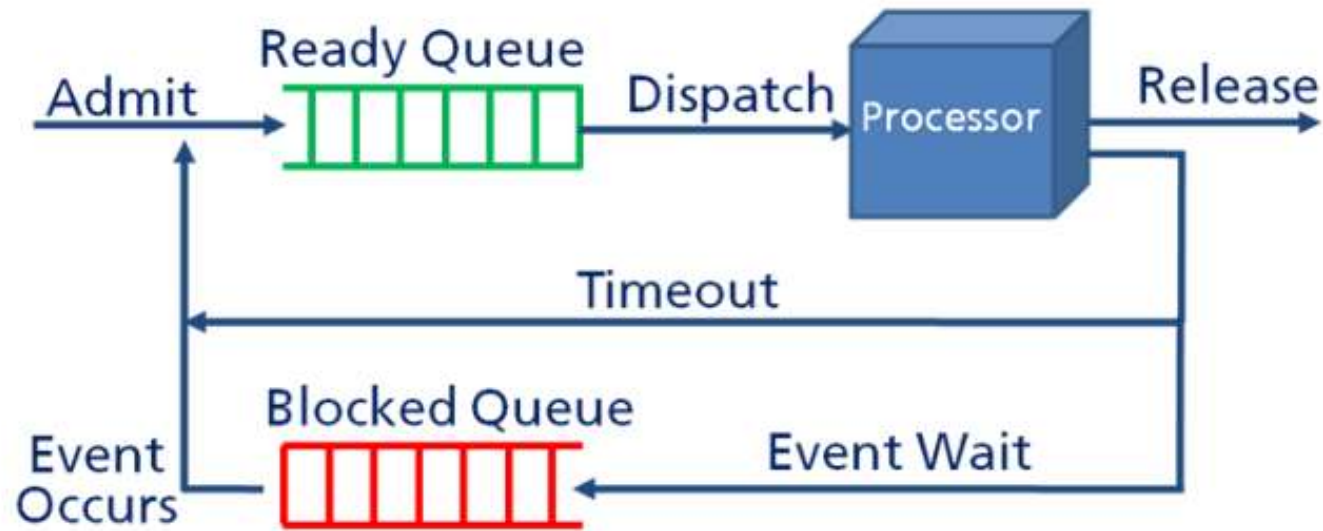
### Example Applications:

Ensuring '*fair treatment*' to each of a list of pending tasks (first come first served)

Simulation: modelling and analysing real-world systems



# Queue Example: Queues in a Process Scheduler of an Operating System



Single Blocked Queue



# Stack ADT

Stack is a *last in, first out* list. No sort order assumed.

Objects stored in stack:

A finite sequence of elements of the **same type**.

Operations:

**n**: Node

**s.push(n)**

**s.pop( )** → **n**

**s.top( )** → **n**

**s.isEmpty( )** → **b**

**s.isFull( )** → **b**

**s**: Stack

Place item **n** on to top of stack

Remove top item from stack and return it

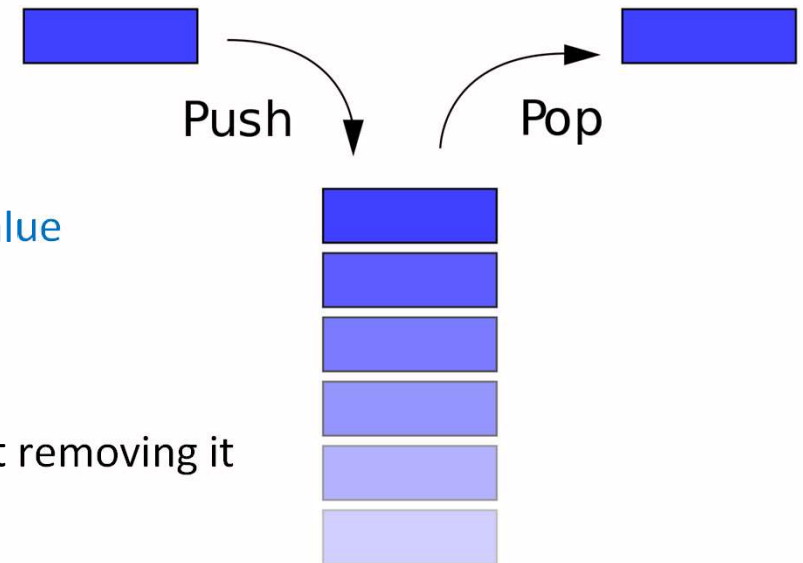
Examine the top item on stack without removing it

Returns **b** = True if stack is empty

Returns **b** = True if stack is full

(relevant if it has limited storage)

**b**: Boolean value



# Stack Interface in Java

Note: Built-in Stack class: `java.util.Stack`

Nonetheless, we will make our own. Why do we bother?

Stack.java:

```
public interface Stack
{
    public void push(Object n);
    public Object pop();
    public Object top();
    public boolean isEmpty();
    public boolean isFull();
}
```

Other operations: `size()`; `makeEmpty()`

Can implement this interface using array, linked list or other storage type



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY



# Stack: Array Implementation (1)

<i>Index</i>	0	1	2	3	4	5	6	7	8	9	10	11
<i>S</i>	18	14	12	10	22	20	19	6				

Consider stack implemented as an array with indexes [0..N-1]

Need to maintain a variable, `numItems`, which counts the number of items in the stack.

`numItems = 0`  $\Rightarrow$  stack is empty

`numItems = N`  $\Rightarrow$  stack is full

Also necessary to know the index of the “top” item and the index of the “bottom” item

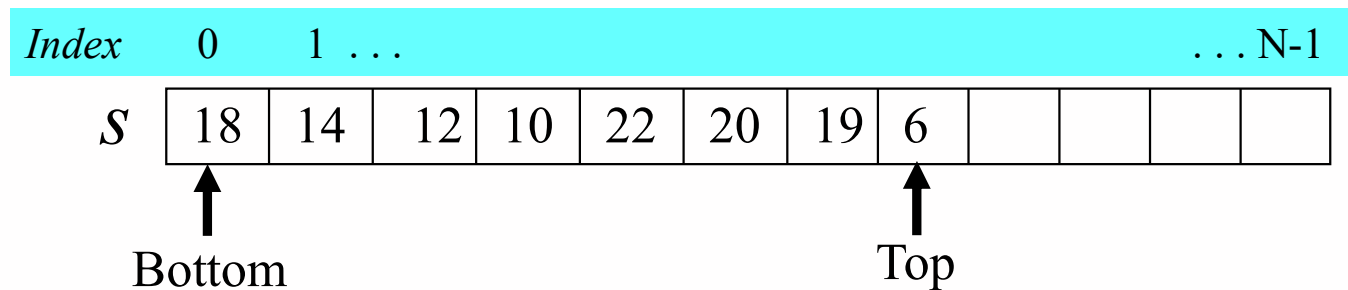
Two possibilities:

**(1)** Top of stack is first index of array

**(2)** Top of stack is last index of array



## Stack: Array Implementation (2)



Of these two possibilities, **(2)** is the more efficient

As both “Push” and “Pop” occur at the top of the stack, it is better to have room at this end.  
The index of Top therefore holds the value  
 $numItems - 1$

With possibility **(1)**, items would have to be shuffled up and down the stack.



# Stack: Array Implementation (3)

Stack operations may be implemented as follows:  
**(basic implementation: no error checking)**

*Push(n)* `top++; s[top] = n; // or: s[++top] = n;`

*Pop()* `n = s[top]; s[top] = null; top--; return n;`

*Top()* `return s[top];`

*isEmpty()* `if (top==-1) return true; else return false;`

*isFull()* `if (top==s.length-1) return true; else return false;`



# Stack: Array Implementation (4)

Fuller implementation:

ArrayStack.java

- Need to make sure before **Push** that it's not already full, and before **Pop/Top** that it's not already empty
- Array has limited capacity, so **isFull()** is required
- Elements stored in **Object** array, so they need to be **cast** to the correct type when popped

Let's look at the code ...



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# Queue ADT

A Queue is a *first-in, first out* list. No sort order assumed.

Objects it stores:

A finite sequence of elements of the same type.  
Front item has been in the queue longest;  
rear item entered the queue most recently.

Operations:

**e**: Element

**q.enqueue(e)**

**q.dequeue( ) → e**

**q.front( )**

**q.isEmpty( ) → b**

**q.isFull( ) → b**

**q**: Queue

Place **e** at rear of **q**, assuming room

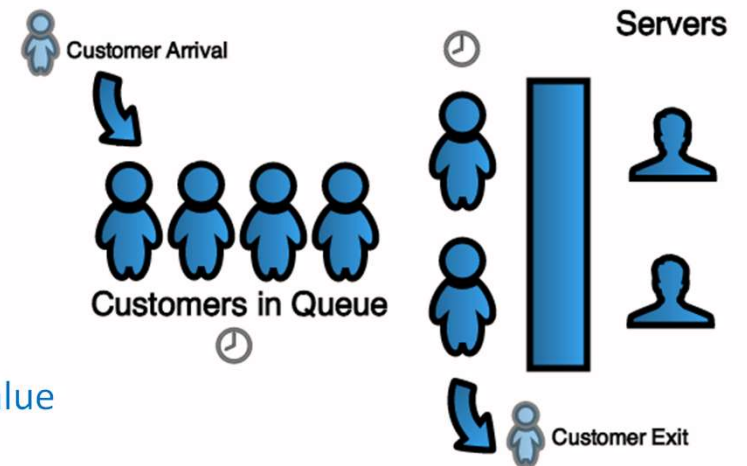
Remove front item from **q** and  
return it

Return front element without removing it

Returns **true** if queue is empty

Returns **true** if queue is full

**b**: Boolean value



# Queue Interface in Java

## Queue.java:

```
public interface Queue
{
    public void enqueue (Object n) ;
    public Object dequeue () ;
    public boolean isEmpty () ;
    public boolean isFull () ;
    public Object front () ;
}
```

Other operations: size(); makeEmpty()

Again, can implement this interface using various storage types

We will use an array: **ArrayQueue.java**

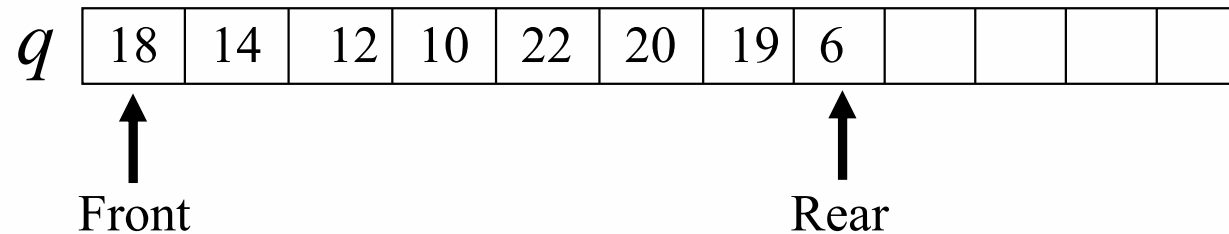
Let's examine it



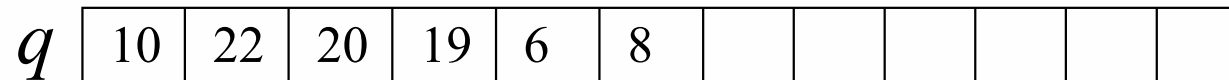
OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Queue: Array Implementation

With array implementation, items must be “shuffled” towards the front after a *dequeue*  
Note that with the array implementation, once *Rear* becomes equal to  $N-1$  no further items can be **enqueued** (array space limitation)



```
n = dequeue( ) ; enqueue (8); n = dequeue( ) ; n = dequeue( ) ;
```





# Any Other Way?

Is there a way of implementing a Queue **as an array** while avoiding shuffles?

What do you think?



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

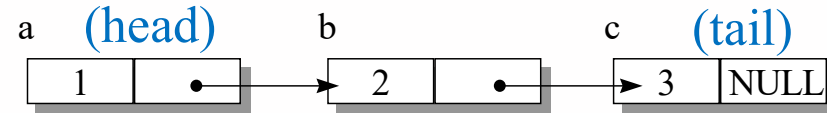
University  
ofGalway.ie

# What You'll Achieve in This Topic

- Discuss the purpose and characteristics of the Linked List ADT
- Show how to implement linked lists in Java
- Develop and explain code to use linked lists and add additional functionality to a linked list ADT
- Describe, implement and use extensions to the basic linked list ADT
- Develop linked list implementations of Stack and Queue ADTs
  - *[You should be able to do this yourself]*

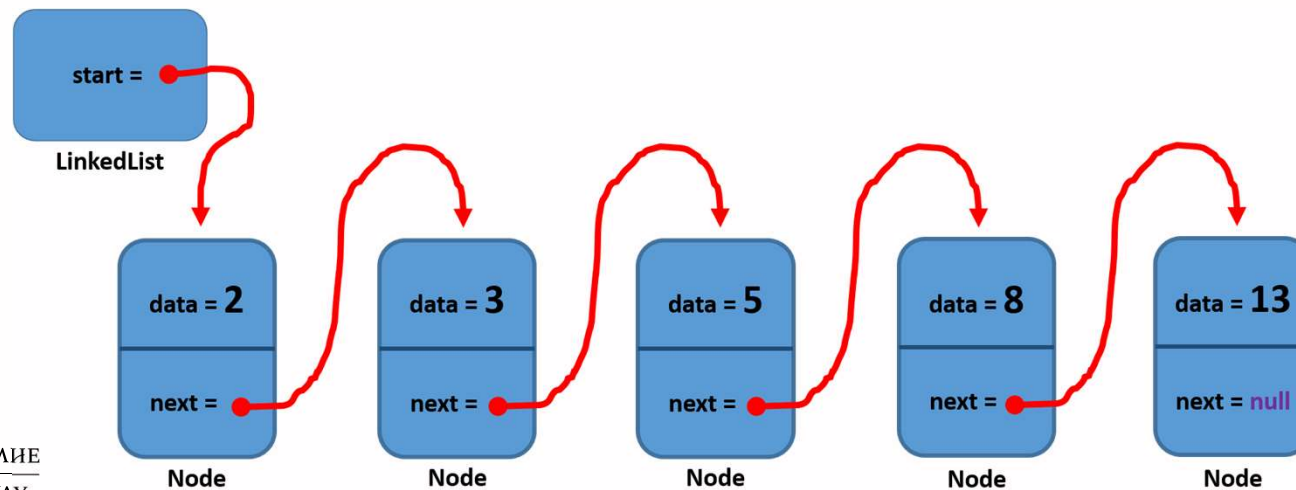


# Linked Lists: Description



## Linear Linked List:

An ADT which stores an arbitrary-length list of data objects as a sequence of nodes  
Each node consists of **data** and a **link** to another node  
Each node, except the last, linked to a successor node



# Linked Lists: Characteristics

- **Self-referential** structure type
  - Every node has a pointer to a node of same type
- Very useful for **dynamically** growing/ shrinking lists of data
- Drastically reduces effort to **add/remove items** from middle of list, when compared to arrays
- Arrays' potential problem of **overflow is solved**
- Sequential access
  - **Inefficient** to retrieve an element at an arbitrary position, relative to array



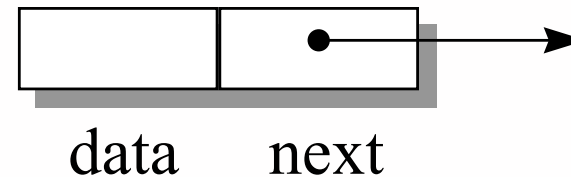
# Linked Lists: Implementation

Define a **Node** class:

Members: **data** (any variables required)  
and **next** (reference to another Node object)

Each node occurrence is linked to a succeeding occurrence by way of the member **next**  
If next is **null**, there is no item after this in list (termed "*tail*" node)

Start-point for the list is the "head" node:  
can trace from it to any other node



# Node for Singly Linked List in Java (1)

```
public class Node {  
    // Instance variables:  
    private Object element;  
    private Node next;  
  
    /** Creates node with null refs to its element and next node. */  
    public Node() {  
        this(null, null);  
    }  
  
    /** Creates node with the given element and next node. */  
    public Node(Object e, Node n) {  
        element = e;  
        next = n;  
    }  
}
```

Node.java on blackboard





## Node for Singly Linked List in Java (2)

```
// Accessor methods:
public Object getElement() {
    return element;
}
public Node getNext() {
    return next;
}

// Modifier methods:
public void setElement(Object newElem) {
    element = newElem;
}
public void setNext(Node newNext) {
    next = newNext;
}
}
```

This is enough to allow us to create nodes and link them, But for a full-fledged ADT we require a LinkedList class to manage them.



# Exercise

## Create some nodes

Store a string in each

Link them in constructors or with `setNext()`

## Iterate through list

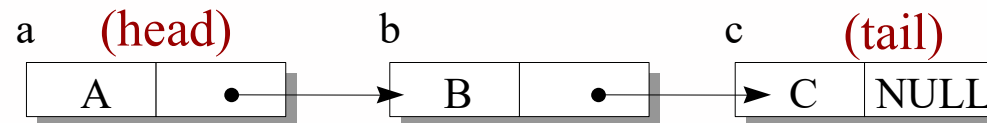
Set a Node reference `currentNode` to point to Head node

Get element data, cast it to a string & display

Set `currentNode` to next node

Loop until end node reached (null)

NodeTest.java on blackboard:  
Incomplete! Will finish in class.



Now build an ADT around these ideas ...



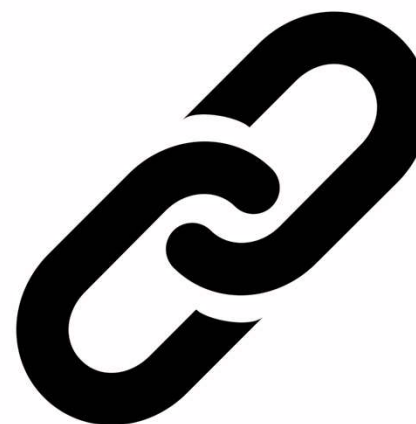
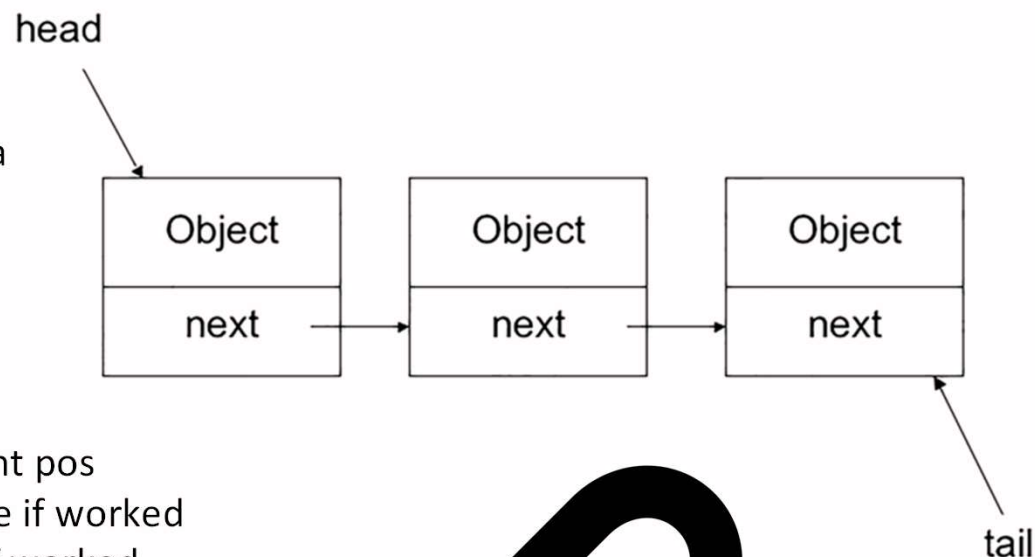
# Linked List ADT

## Definitions of methods to manage Linked List

Don't create nodes manually, just supply element data  
Keeps track of current position in list

## Typical methods in a Linked List ADT

long <b>size</b> ( )	Returns size of list
boolean <b>isEmpty</b> ( )	True if list is empty
Object <b>getCurr</b> ( )	Returns element at current pos
boolean <b>gotoHead</b> ( )	Sets curr pos to head; true if worked
boolean <b>gotoNext</b> ( )	Moves to next pos; true if worked
<b>insertNext</b> (Object el)	Creates new node after current
<b>deleteNext</b> ( )	Removes the node after current
<b>insertHead</b> (Object el)	Creates a new node at head
<b>deleteHead</b> ( )	Removes the head node



# Singly Linked List Class

Stores the head of the list and current position

For efficiency, also keeps track of current size

(could alternatively just count its nodes when needed)

```
public class SLinkedList {
    protected Node head;    // head node of the list
    protected Node curr;    // current position in list
    protected long size;    // number of nodes in the list

    /*( Default constructor that creates an empty list */
    public SLinkedList() {
        curr = head = null;
        size = 0;
    }
    // ... Insert, remove and search methods go here ...
}
```



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

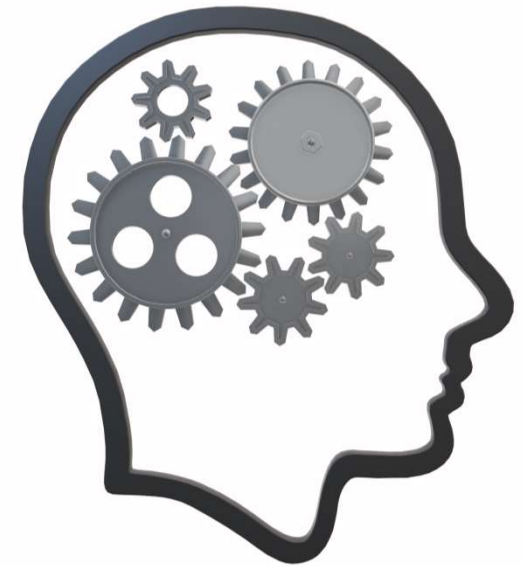
# What You'll Achieve in This Topic

- Explain the purpose of algorithm analysis
- Describe  $O$ -notation and functions used with it, as well as related notations
- Analyse the space and time requirements of any algorithm encountered in this module
- Interpret the results of complexity analysis in terms of whether an algorithm is efficient or not
- Compare algorithms and make rational choices about which is best for an application
- Explain the concept of Dynamic Programming, show simple applications of it, and analyse its effects
- Describe P, NP and NP-complete problems and related terminology



# Motivation

- There is more than one way of doing everything
  - Which algorithm should I choose for a given job?
- All algorithms take CPU **time** and **memory** space
  - Often can make tradeoffs: choose algorithm variants that either use more memory or more CPU
  - If memory space requirements are large, program may use disk swap space rather than RAM: much slower!
  - Memory requirements very large: program cannot run
  - Often identify the algorithms that don't require "too much" space, then choose the one with lowest CPU
- Need to compare algorithms' time & space requirements
  - Purpose of algorithm analysis



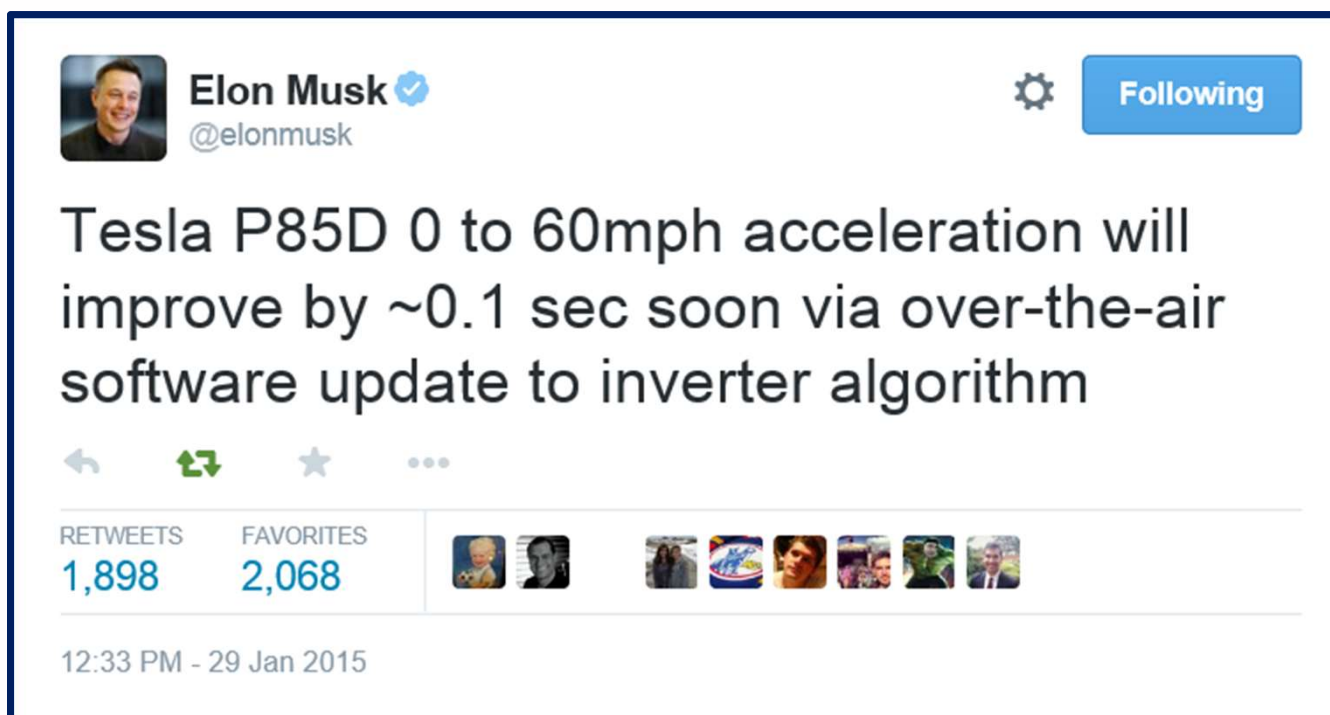
# Value of Algorithm Optimisation...



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY



# Value of Algorithm Optimisation...



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

More recent story:

<https://www.theverge.com/2018/6/2/17413732/tesla-over-the-air-software-updates-brakes>

# Algorithm Analysis Basics

## Why not just run it to measure time & space used?

Sometimes done when theoretical analysis difficult

## Want to be able to ...

Evaluate algorithms “on paper”, *without* having to first implement, debug and test them all

Have a measurement that’s *independent* of particular computer configuration

Compare algorithms reliably, without being influenced by variations in implementation

Understand how it will perform on large problems

Identify “hot spots” to give our attention to, when developing and optimising programs



# Algorithm Analysis Basics (1)

## Theoretical Analysis:

Uses a high-level **pseudocode** description of the algorithm instead of an implementation  
Characterises run-time as a function of input size, **n**

**What is n for a sorting algorithm?**

This function specifies the **order of growth** rate of runtime as n increases

Takes into account all possible inputs

Evaluates speed independent of hardware/software



# Algorithm Analysis Basics (2)

Basic approach: derive function for the:  
**count of the primitive operations**

- These are the individual steps performed by program
- Assume each step takes the same time
- Examine any terms that control repetition



# Counting Primitive Operations (1)

	<i># operations</i>
<b>Algorithm</b> <i>arrayMax</i> ( <i>A</i> , <i>n</i> )	
<i>currentMax</i> = <i>A</i> [0]	2
for <i>i</i> = 1 to <i>n</i> - 1 do	2 <i>n</i>
if <i>A</i> [ <i>i</i> ] > <i>currentMax</i> then	2( <i>n</i> - 1)
<i>currentMax</i> = <i>A</i> [ <i>i</i> ]	2( <i>n</i> - 1)
{increment counter <i>i</i> }	2( <i>n</i> - 1)
return <i>currentMax</i>	1
	Total: 8 <i>n</i> - 3

## Example:

Algorithm to find largest element of an array

Count max number of operations as fn of array size, **n**



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

Note: Some textbooks/websites get different answers for this example. **Why?**

School of Computer Science

# Counting Primitive Operations (2)

- Could consider **average**, **best** or **worst** case  
Consider searching for a given value in an array:  
what are average, best and worst case no. of ops?  
What about finding the largest value in an array?  
What about sorting an array?
- Usually analyse worst case  
Want algorithms to work well even in bad cases  
Average case important too, if different from worst case
- These counts are basis of O Notation  
“big-oh notation”  
[Greek letter Omicron looks same as Latin letter O]



# O Notation

## Approach:

Derive expression for count of basic operations (as discussed)

Focus on dominant term, ignore constants

E.g.  $O(5n^2 + 1000n - 3) \Rightarrow O(n^2)$

Since constant factors and lower-order terms are eventually dropped, can disregard them when counting primitive operations

## Example:

Algorithm arrayMax runs in \_\_\_\_ time

Or, “arrayMax has run-time complexity of order \_\_\_\_”



# O Notation: Details

Used for **Asymptotic Analysis of Complexity**

Trend in algorithm's run time as  $n$  gets large

Look at **Order of Magnitude** of no. of actions

Independent of computer/compiler/etc

Note: specifically care about **tightest** upper bound

An algorithm that is  $O(n^2)$  is also  $O(n^3)$ , but the former is more useful





# O Notation: Details

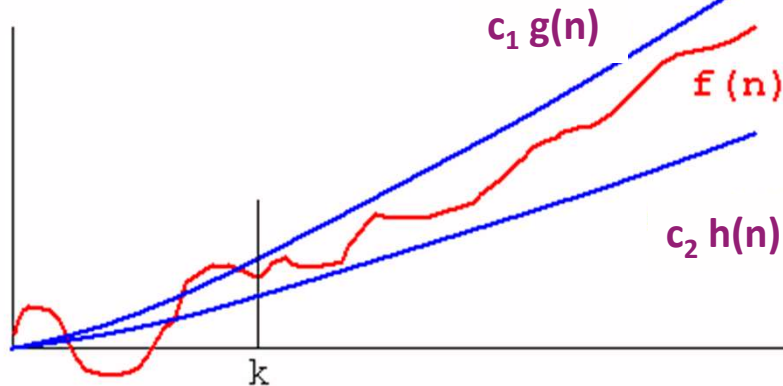
The function specified in O notation is the **upper bound** on the behaviour of the function (algorithm) being analysed

Can be best-case / average-case / worst-case behaviour

Example: if  $f(n)$  is the algorithmic function and we deduce that  $f(n)$  is  $O(g(n))$ , this means that as  $n \rightarrow \infty$ ,  $f(n) \leq c \cdot g(n)$

Function in  $O()$

Some constant



$c_1 g(n)$  is bound specified by O notation

$c_2 h(n)$  is bound specified by  $\Omega$  notation

If  $g(n) = h(n)$ ,  $c_1$  and  $c_2$  specify bound in  $\Theta$  notation



# O Notation: Example

Example of Big O notation:

$$\text{Let } f(x) = 6x^4 - 2x^3 + 5$$

Apply the following rules:

- If  $f(x)$  is a sum of several terms, only the one with the largest growth rate is kept
- If  $f(x)$  is a product of several factors, any constants that do not depend on  $x$  are omitted.
- Thus, we say that  $f(x)$  has "big-oh" of  $(x^4)$ .
- We can write  $f(x)$  is  $O(x^4)$



# Important Functions Used in O Notation

Functions commonly used:

Constant:	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
n-Log-n	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

Will compare graphically ...

Notes:

Convention: logs are base 2 since not otherwise stated

Just because two algorithms have same complexity does not mean they take *exact* same time;

It means their running times will be *proportional*.



# Comparisons of Functions (1)

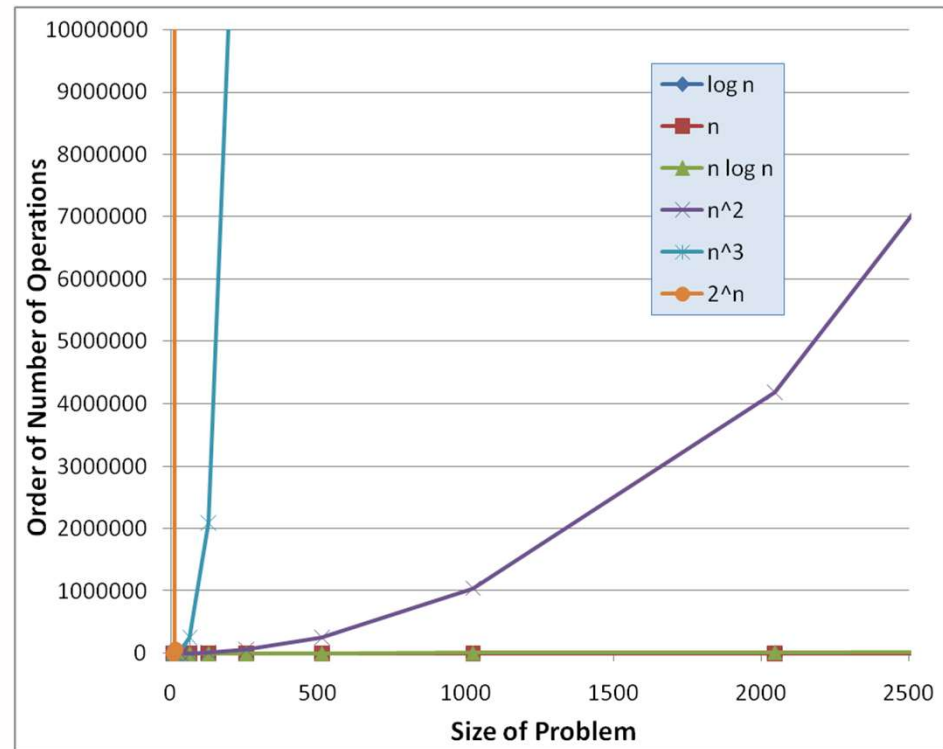
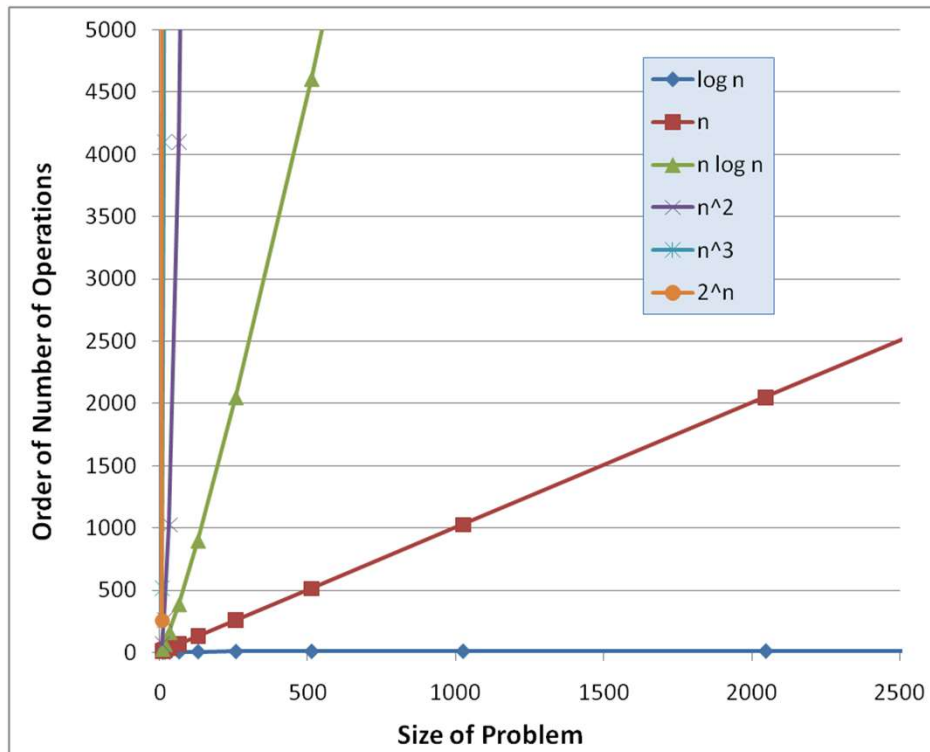
n	const.	log n	n	n log n	n <sup>2</sup>	n <sup>3</sup>	2 <sup>n</sup>
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65536
32	1	5	32	160	1024	32768	4294967296
64	1	6	64	384	4096	262144	1.84467E+19
128	1	7	128	896	16384	2097152	3.40282E+38
256	1	8	256	2048	65536	16777216	1.15792E+77
512	1	9	512	4608	262144	1.34E+08	1.3408E+154

<http://bigocheatsheet.com/>

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
<u>Array</u>	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
<u>Stack</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Queue</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Singly-Linked List</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Doubly-Linked List</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$

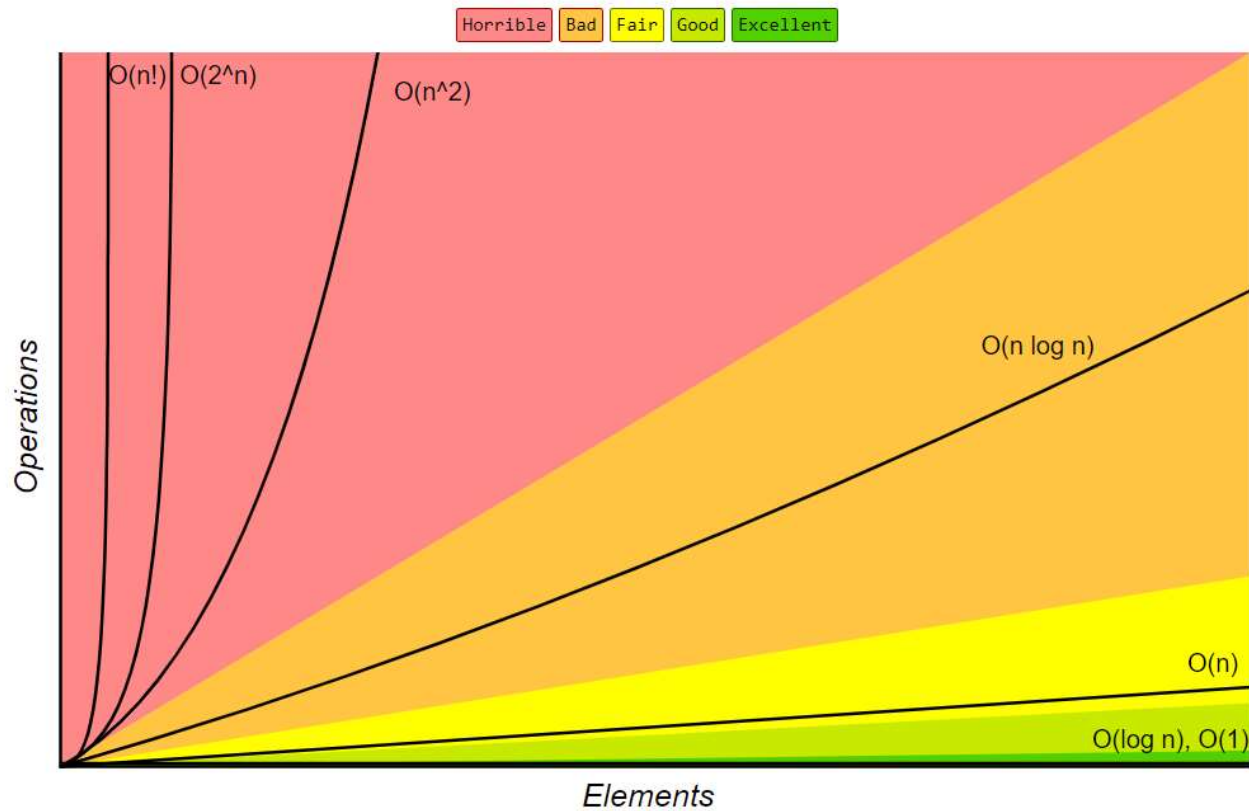


# Comparisons of Functions (2)



# Comparisons of Functions (3)

Big-O Complexity Chart



# Some Intuitions

What kind of operations would have complexity of:

$O(n)$ ?

$O(n^2)$ ?

$O(\log n)$ ?

Will do examples in class...



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# Efficiency and O Notation

- **Constant**
  - Most efficient possible, but only applicable to simple jobs
- **Logarithmic, Linear, and  $n \log n$** 
  - If an algorithm is described as “efficient”, this usually means  **$O(n \log n)$**  or better
- **Quadratic and Cubic**
  - Not very efficient, but polynomial algs. usually considered “tractable”: acceptable for problems of reasonable size
- **Exponential**
  - Very inefficient:  
problems that (provably) require an algorithm greater than polynomial complexity are “hard”







OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

# Review of Recursion (1)

Methods can call others, but can also call themselves

Directly; or indirectly via another method

This creates a form of loop: termed **recursion**

Like other loops, recursive methods have to be carefully constructed so that they **terminate** correctly

Why?

Small but significant set of problems can be solved elegantly in this way (e.g. in maths, list processing)

Know the solution to simple **base cases**

More complex cases are defined in terms of base cases

Example: Fibonacci sequence:

$f(0) \rightarrow 0$   $f(1) \rightarrow 1$

$f(n) \rightarrow f(n-1) + f(n-2)$ , where  $n > 1$



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

See: [Fibonacci.java](#)

School of Computer Science

# Review of Recursion (2)

Iteration can be used anywhere you can use recursion

Sometimes recursion is **more elegant**, if it reflects the way the problem is usually thought about

Aim to use most **intuitive** representation of problems

**Complexity analysis** can be easier in some cases

Drawbacks of recursion:

**Inefficient** use of function heap: large amount of deeply nested method calls to be kept track of together

If done naively, **number of calls** can explode:

f(20) takes 21,891 calls to method

f(30) takes 2,692,537 calls

Easier to avoid these problems with normal loops

Depending on algorithm, need to take care not to recompute values unnecessarily: [see next](#)



# Review of Recursion (3)

A recursive method must have:

1. **Test** to stop or continue the recursion
2. An **end case** that terminates the recursion
3. **Recursive call(s)** that continue the recursion

Use recursion if:

Recursive solution is natural and easy to understand

Recursive solution does not result in excessive duplicate computation

Equivalent iterative solution is too complex

If performance is an issue ...

Perhaps use recursion to write the first version

Later re-write using less natural but more efficient iteration



# Avoiding Unnecessary Computations

Fibonacci program:

Could store interim results rather than computing them from scratch every time

Could use an array:

At each index  $i$ , store answer for fibonacci( $i$ )

Initialise first 2 values to base cases

Initialise all others to "don't know" values (-1)

When calling method, check to see if we know the answer before computing it

Will this make a difference?

Will do a formal analysis in a few minutes ...



# Dynamic Programming

Fibonacci with storage is an instance  
of **Dynamic Programming**

Richard Bellman, 1940s

Basic idea:

Solve complex problem by breaking it  
into simpler sub-problems

When solution to a sub-problem is found,  
store it ("memo-ize") so it can be re-used without recomputing it  
Combine the solutions to sub-problems to get the overall solution  
Particularly useful when the number of repeating  
sub-problems **grows exponentially** with problem size



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Dynamic Programming

In general, takes problems that appear exponential, produces polynomial-time algorithms for them

Will see this in analysis

Trade-off of *storage* and *speed*

Widely used in heuristic optimisation problems

Examples of DynProg algorithms:

Finding longest common sub-sequence in two strings

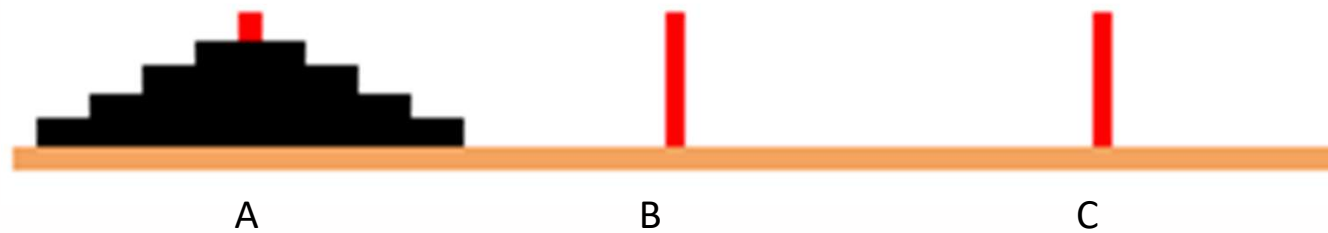
<https://www.cs.usfca.edu/~galles/visualization/DPLCS.html>

Towers of Hanoi

Knapsack problem



# Towers of Hanoi



**Walkthrough:**

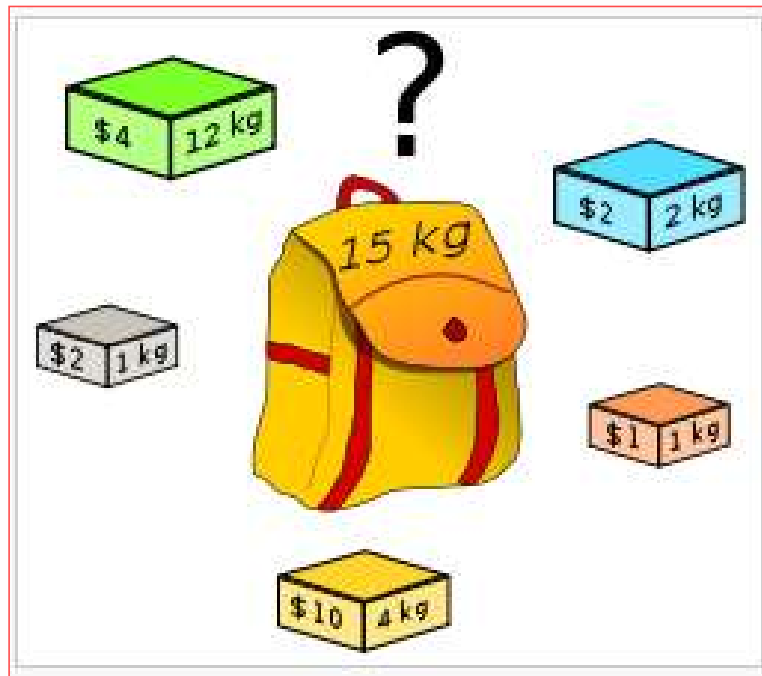
<http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/13/hanoi.html>



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY



# Knapsack Problem [Wikipedia]



Example of a one-dimensional (constraint) knapsack problem: which boxes should be chosen to maximize the amount of money while still keeping the overall weight under or equal to 15 kg? A **multiple constrained problem** could consider both the weight and volume of the boxes. (Solution: if any number of each box is available, then three yellow boxes and three grey boxes; if only the shown boxes are available, then all but the green box.)



Also see: <https://www.dyclassroom.com/dynamic-programming/0-1-knapsack-problem>  
OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

# Dynamic Programming

Problem structure requires 3 components:

1. **Simple sub-problems:** must be able to break overall problem into indexed sub-problems & sub-sub-problems
2. **Sub-problem decomposition:** Optimal/correct solution to overall problem must be composed from sub-problems
3. **Sub-problem overlap:** so elements can be re-used



# Dynamic Programming

## Basic Steps in DynProg Approach:

1. Set up the overall problem as one decomposable into overlapping sub-problems that can be indexed
2. Solve sub-problems as they arise;  
**store solutions** in a table
3. Derive overall solution from the solutions in the table



# Dynamic Programming – A Note

Don't get confused!

Dynamic Programming  $\neq$  Java Programming Language's property of using dynamic memory allocation

Meaning of "Dynamic Programming"

Here, a "programme" is an *optimised plan* (sequence of steps)

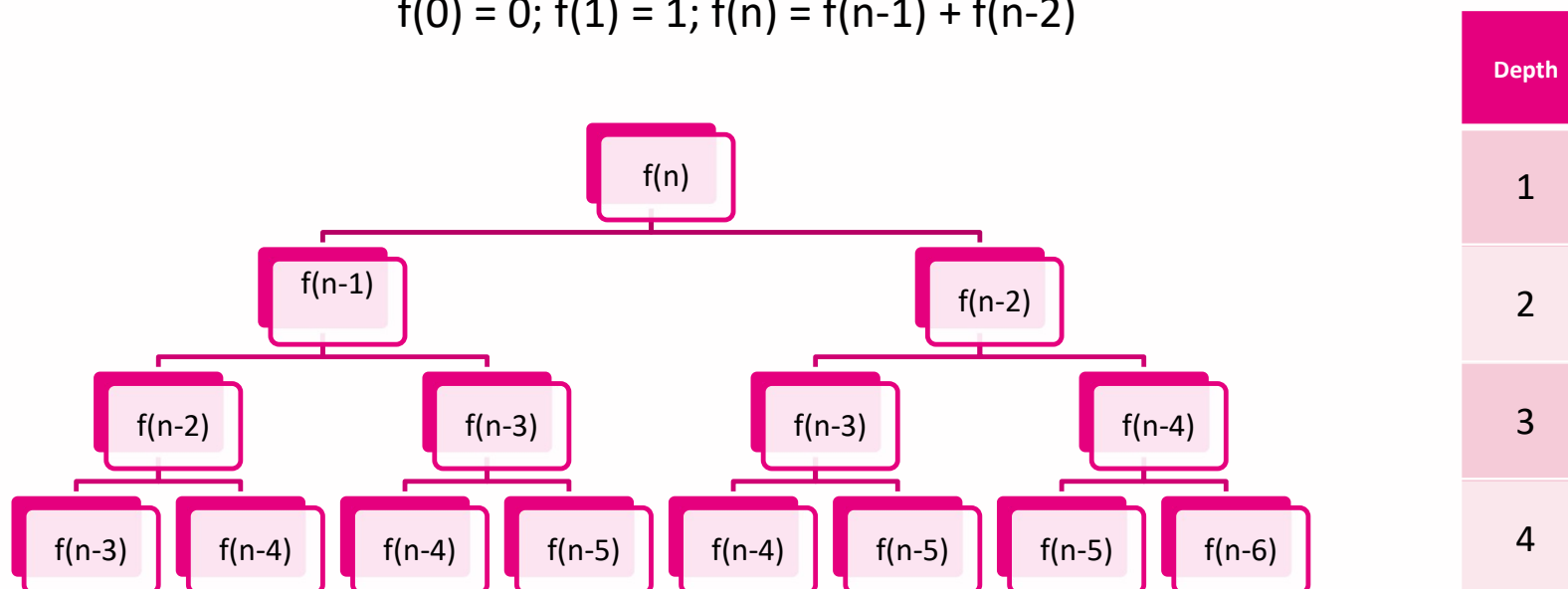
"Dynamic" because the "memo" of past results (table storing them) is updated as procedure progresses

In contrast to a procedure where you calculate a fixed set of "tables" and then use them (e.g. log tables)



# Example: Fibonacci Sequence

$$f(0) = 0; f(1) = 1; f(n) = f(n-1) + f(n-2)$$

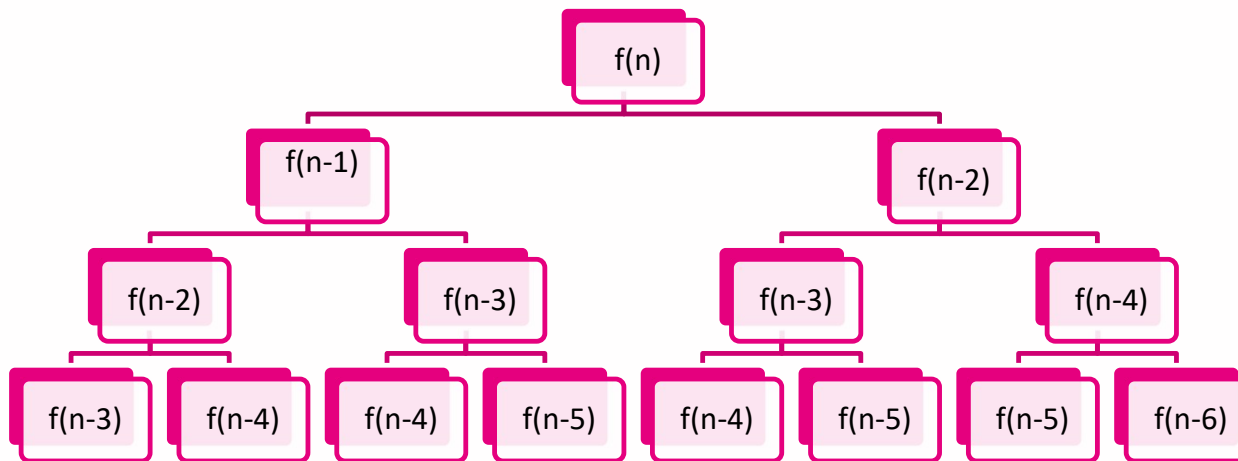


In class: compute complexity for both cases,  
without and with Dynamic Programming



# Example: Fibonacci Sequence

$$f(0) = 0; f(1) = 1; f(n) = f(n-1) + f(n-2)$$



Depth	#Evals
1	$2^0$
2	$2^1$
3	$2^2$
4	$2^3$

- If we **don't** store values, overall complexity determined by no of evaluations at bottom level
- If we **do** store values, 'new' calculations are down left side: this is **Dynamic Programming** approach



# More Big Greek Letters (1)

$O(n \log n)$ : "Big Oh" (Omicron)

Upper bound on of asymptotic complexity

In this case: there is a constant  $c_2$  s.th.  $c_2 n \log n$  is an upper bound on asymptotic complexity

$\Omega(n \log n)$ : "Big Omega"

Specifies lower bound on asymptotic complexity

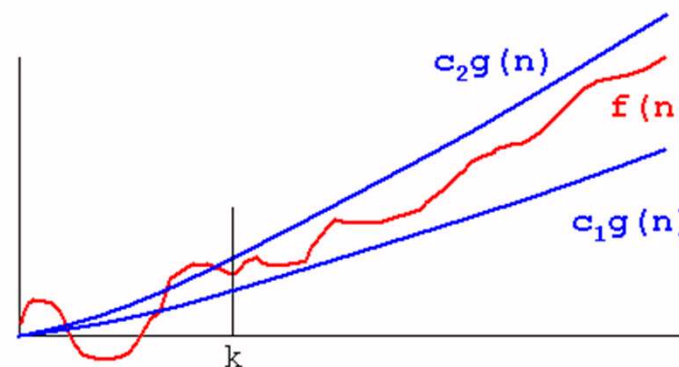
In this case, algorithm has a lower bound of  $c_1 n \log n$

$\Theta(n \log n)$  "Big Theta"

Specifies upper and lower bounds

In this case, there exist 2 constants,  $c_1$  &  $c_2$  s.th.

$c_1 n \log n < f(n) < c_2 n \log n$



Don't confuse upper/lower bounds with best/worst case: all cases have all bounds



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

# More Big Greek Letters (2)

Of these,  $\Theta()$  makes the strongest claims

It specifies that the growth rate is no better and no worse than some level

Requires additional analysis relative to  $O$

There are also some others ...

These are common in Maths but not Comp Sci

**Little o:**  $o(g(n))$  specifies a function  $g(n)$  that grows much *faster* than the one we are analysing

**Little omega:**  $\omega(g(n))$  specifies a function  $g(n)$  that grows much *slower* than the one we are analysing





# P, NP, and NP-Complete Problems [1]

**P Problems** are those for which:

There is a **deterministic** algorithm that solves it in **Polynomial Time**

In other words, algorithm's complexity is  $O(p(n))$  where  $p(n)$  is a polynomial function

Note: Problems that can be solved in polynomial time are termed **tractable**

All worse are termed **intractable**



# P, NP, and NP-Complete Problems [2]

NP Problems are **Nondeterministic Polynomial** ...

**Nondeterministic** algorithms have 2 repeating steps:

Generate a *potential* solution, randomly or systematically

Verify whether it is right

If not, repeat

If the verification step is **polynomial**, the algorithm and associated problem are NP

E.g. factoring large integers as used in RSA encryption

Subset sum problem: (next slide)

Note that P is a subset of NP:

Using NP approach, generate a guess, then in the verification step ignore it and solve it as usual



# Examples of P and NP Problems

(Trivial) example of a P problem:

Search an array of integers for a certain value

Example of an NP problem: Subset problem

Given a set of integers, does some nonempty subset of them sum to 0

E.g. does a subset of the set  $\{-2, -3, 15, 14, 7, -10\}$  add up to 0?

There is no polynomial algorithm to solve this problem

However, verification of a potential solution is polynomial,  
in fact  $O(n)$  (just add up the numbers in the potential solution)

Example of an NP problem: Integer Factorisation



# P, NP, and NP-Complete Problems [3]

**NP-Complete** Problems are those that are "as hard as all others" in NP

Algorithms that are comparable to ("polynomially reducible to") others in NP but not reducible to P

**Polynomially reducible:** there is some polynomial-time transformation that converts the inputs for Problem X to inputs for Problem Y

**NP-Hard** Problems are those that are "as hard or harder than" all in NP

A problem X is NP-Hard if NP-Complete problems are polynomially reducible to it



# NP-Complete Problems

NP-Complete is a complexity class which represents the set of all problems  $X$  in NP for which it is possible to reduce any other NP problem  $Y$  to  $X$  in polynomial time.

Intuitively this means that we can solve  $Y$  quickly if we know how to solve  $X$  quickly.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time.



# NP-Hard Problems

Intuitively, these are the problems that are at least as hard as the NP-complete problems.

Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

The precise definition here is that “a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time”

Example:

The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt?



# The “P versus NP” Problem

Major unsolved problem in computer science

“If the solution to a problem is easy to check for correctness, is the problem easy to solve?”

“whether every problem whose solution can be **quickly verified** by a computer can also be **quickly solved** by a computer”

**Quickly** means there exists an algorithm to solve the task that runs in *polynomial time*

P (polynomial) ← solvable in polynomial time

NP (non-deterministic polynomial) ← only verifiable in polynomial time



# The “P versus NP” Problem

An answer to the  $P = NP$  question will determine whether all problems that can be verified in polynomial time can also be solved in polynomial time.

If it turned out that  $P \neq NP$ , it would mean that there are problems in NP (such as *NP-complete problems*) that are harder to compute than to verify.

We already know that  $P \subseteq NP$ !

In theoretical computer science the problems considered for P and NP are decision problems

Decision problems don't provide numeric results, but yes/no answers





# P, NP, and NP-Complete Problems

## Is P=NP?

Are there deterministic polynomial solutions to all NP problems, that we just have not found yet?

One of the current grand challenges in Computer Science

Prizes offered, etc.

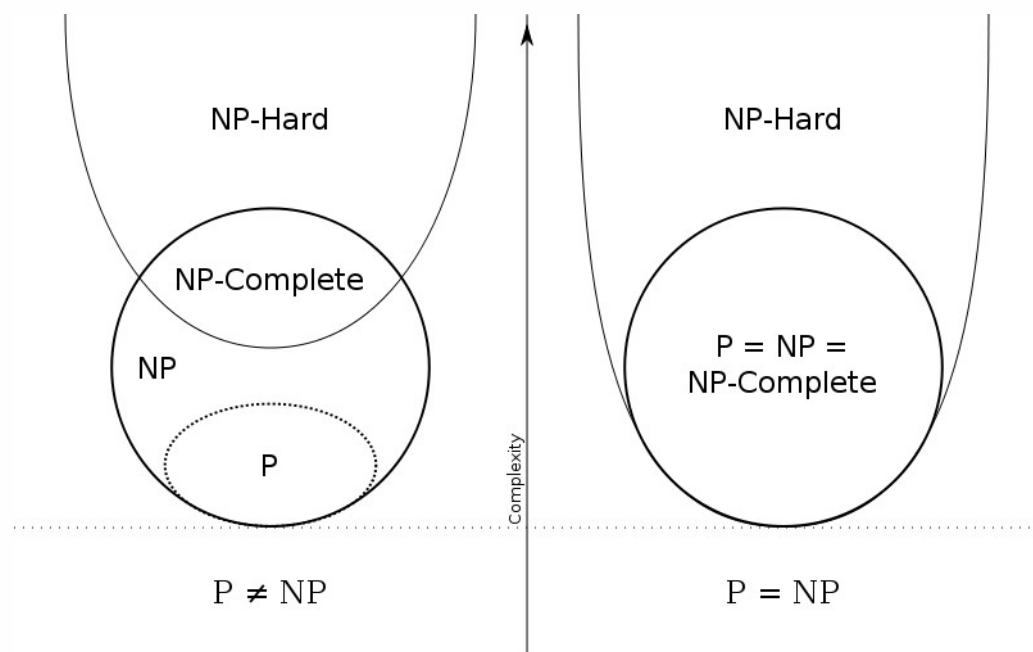


Image: [Wikimedia Commons](#)



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

# Learning Objectives Review

- Now that you have completed this topic successfully, you should be able to:
- Explain the purpose of algorithm analysis
- Describe  $O$ -notation and functions used with it, as well as related notations
- Analyse the space and time requirements of any algorithm encountered in CT2109
- Interpret the results of complexity analysis in terms of whether an algorithm is efficient or not
- Compare algorithms and make rational choices about which is best for an application
- Explain the concept of Dynamic Programming, show simple applications of it, and analyse its effects
- Describe P, NP and NP-complete problems and related terminology



# Further Reading for Revision

## Big O Notation

<https://www.bigocheatsheet.com/>

<https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/>

<https://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html>

## P, NP, NP-Hard and NP-Complete

<https://towardsdatascience.com/the-aged-p-versus-np-problem-91c2bd5dce23>

<https://bigthink.com/technology-innovation/what-is-p-vs-np>



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

School of Computer Science



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

# What You Will Achieve in This Topic

- Analyse the complexity of Sequential and Binary Search
- Describe the general operation and algorithm details of a variety of sorting algorithms
- Implement and use these algorithms
- Analyse the algorithmic complexity of the algorithms
- Discuss the relative merits of different sorting algorithms and select the most appropriate for a specific task



# Complexity Analysis of Searching

- Sequential Search:  
Loop through items one at a time  
*What is its complexity?*
- Binary Search:  
"Think of a number between 1 and 500"  
In each iteration, halve the search space  
*What is its complexity?*



**Complexity:** Suppose you have 128 items: cut to 64, 32, 16, 8, 4, 2, 1. That's  $2^0, 2^1, 2^2, 2^3 \dots 2^7$ . So  $\log_2(128) = 7$  (by definition of logs). So, list of 128 items takes 7 steps; list of N items takes  $\log_2(N)$  steps.

Does Binary Search impose any requirements?



# Overview of Sorting

Many **comparison-based** algorithms exist:

- Naive Sort
- Bubble Sort
- Selection Sort
- Insertion Sort
- Shell Sort
- Merge Sort
- Quick Sort

All based on comparisons & swapping:

```
temp = a[i];  
a[i] = a[j];  
a[j] = temp;
```

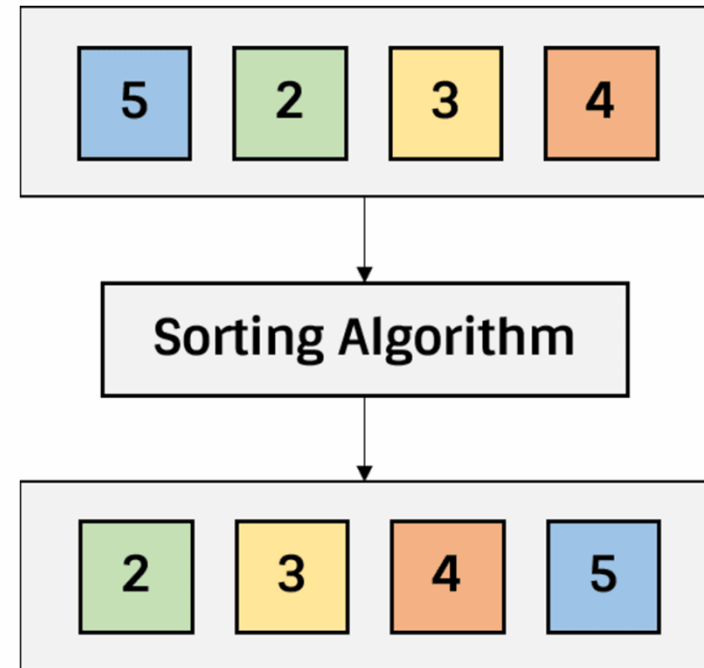
Alternative: address-calculation methods

- Specialised but more efficient

## Sorting in Java Standard API



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY



# Naïve Sort

Naive.java

Probably the world's worst sorting algorithm!

a.k.a. Bogosort

Given an unsorted array:

Shuffle it

Test to see if it's in the right order

If not, repeat

Complexity:

Will do in class ...

Hint: much worse than exponential complexity!



OLLSCOIL NA GAILLIMH  
UNIVERSITY OF GALWAY



# Sorting Concept: Keys & Values

Each object to be sorted can be considered to have a key and a value

Key is what we sort by, value just follows

Example:

Student has properties Name, ID, Grade

Sort by ID  $\Rightarrow$  ID is key and name & grade are values

When swapping two items, swap everything, not just IDs!

Could alternatively use a different property as key

In simplest case, have keys but no other values

E.g. Array of ints



# Java Interface: Comparator

Compares two objects to say which should come first

In Java, any class that implements `java.util.Comparator` interface  
Just one method required to be implemented:

`int compare(Object ob1, Object ob2)`

Negative  $\Rightarrow$  `ob1 < ob2`   0  $\Rightarrow$  `ob1 == ob2`   Positive  $\Rightarrow$  `ob1 > ob2`

```
class Sortbyname implements Comparator<Student> {  
  
    // Method  
    // Sorting in ascending order of name  
    public int compare(Student a, Student b)  
    {  
  
        return a.name.compareTo(b.name);  
    }  
}  
  
//https://www.geeksforgeeks.org/comparator-interface-java/
```



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# Java Interface: Comparable

For a given object, compare it with another to see which should come first

The two objects' class must implement `java.lang.Comparable`

This has just one method to implement:

`int compareTo(Object other)`

Standard classes such as **String** implement this

```
import java.io.*;
import java.util.*;
```

```
class Pair implements Comparable<Pair> {
    String firstName;
    String lastName;

    public Pair(String x, String y)
    {
        this.firstName = x;
        this.lastName = y;
    }

    public String toString()
    {
        return "(" + firstName + " , " + lastName + ")";
    }

    @Override public int compareTo(Pair a)
    {
        // if the string are not equal
        if (this.firstName.compareTo(a.firstName) != 0) {
            return this.firstName.compareTo(a.firstName);
        }
        else {
            // we compare lastName if firstNames are equal
            return this.lastName.compareTo(a.lastName);
        }
    }
}
//https://www.geeksforgeeks.org/comparable-interface-in-java-with-examples/
```



# Insertion Sort (1)

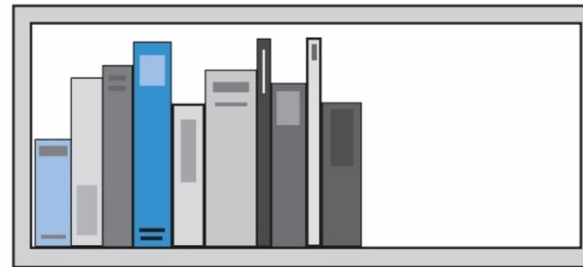
Consider sorting  
a bookshelf ...

If first two books are  
out of order

Remove second book  
Slide first book to right  
Insert removed book into first slot

Then look at third book, if it is out of order

Remove that book  
Slide 2<sup>nd</sup> book to right  
Insert removed book into 2<sup>nd</sup> slot

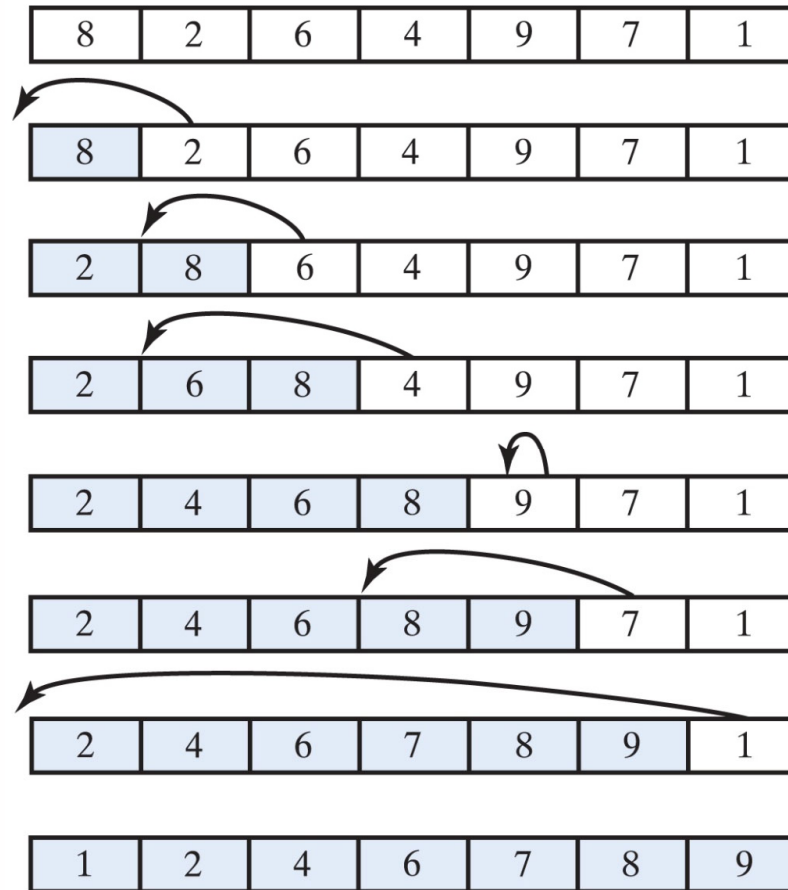


1. Remove the next unsorted book.
2. Slide the sorted books to the right one by one until you find the right spot for the removed book.
3. Insert the book into its new position.



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Insertion Sort Example



# Insertion Sort Algorithm in Pseudocode

Array is indexed from 0  
=> Start at index 1 when  
searching backwards  
Also, First = 0

```
FOR ToSort = 1 to N-1 STEP 1
  SET Index = ToSort-1; SET ToSortEl = A[ToSort]
  WHILE (Index >= First) AND (A[Index] > ToSortEl)
    A[Index+1] = A[Index] // shuffle elements to right
    Index = Index - 1
  END WHILE
  A[Index+1] = ToSortEl // insert element to sort in its place
END FOR
```



# Efficiency of Insertion Sort

Implementation of algorithm:

**InsertSort.java**

Analysis: will do in class

Best case efficiency is  $O(n)$

Worst case efficiency is  $O(n^2)$

If array is closer to sorted order

Insertion sort does less work

Operation is more efficient

This leads to a related algorithm, Shell Sort ...



# Shell Sort (1)

More efficient than Selection Sort or Insertion Sort

Compares distant items first and works way down to nearby items

Interval is called the **gap**

Gap begins at one-half the length of the list and is successively halved until each item compared with neighbour

Motivation:

Insertion Sort tests elements in adjacent locations: may take several steps to get to final location

It is more efficient if array is partially sorted

By making larger jumps, array becomes “*more sorted*” more quickly





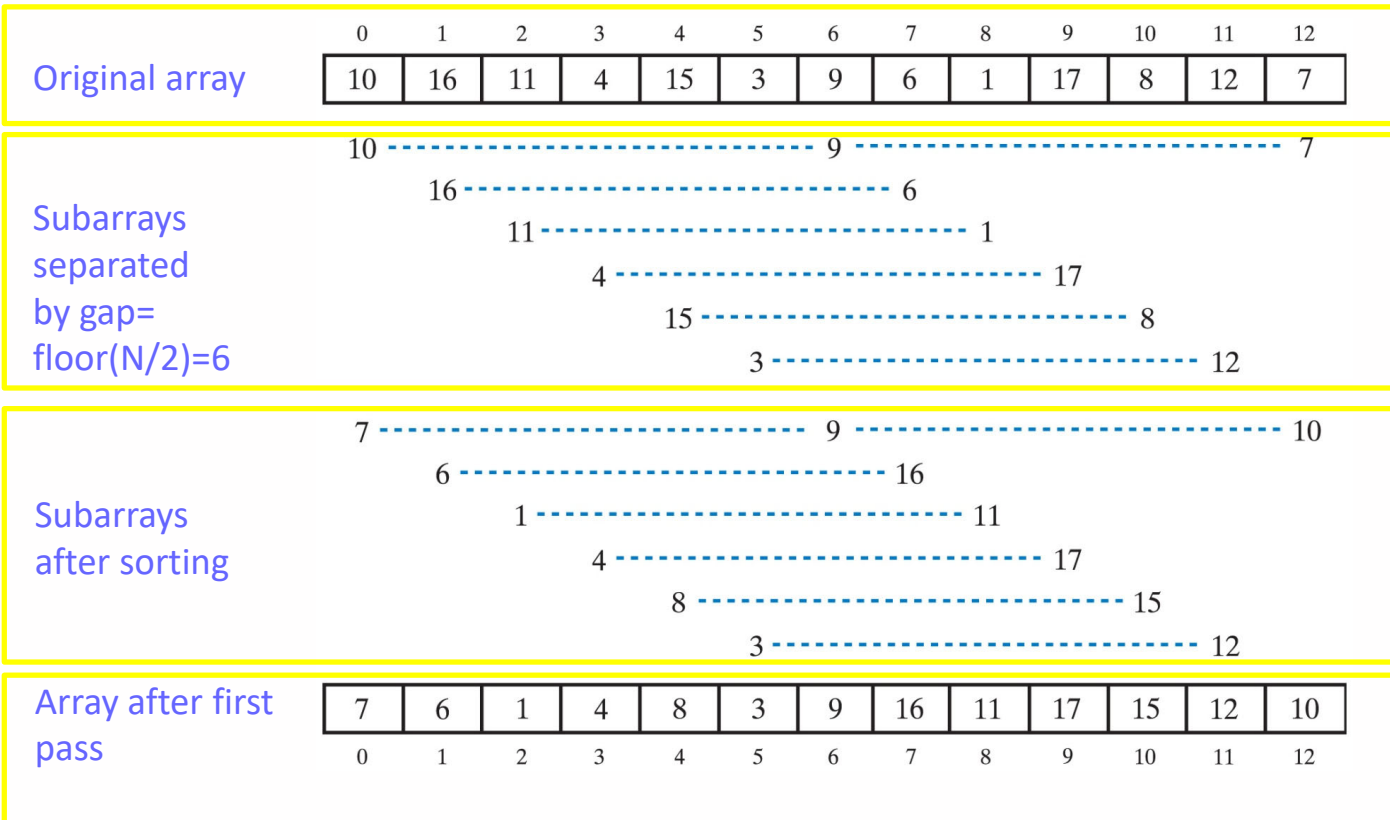
# Shell Sort Algorithm in Pseudocode

```
INPUT: A[0 .. N-1]
SET Gap = N/2, Round to nearest odd no [as it's best with odd-size gap]
WHILE Gap >= 1
  FOR First=0 to (Gap-1) // insertion sort follows
    FOR ToSort = (First+Gap) to N-1 step Gap
      SET Index = ToSort - Gap; SET ToSortEl = A[ToSort]
      WHILE (Index >= First) AND (A[Index] > ToSortEl)
        A[Index+Gap] = A[Index] // shuffle elements to right
        Index = Index - Gap
      END WHILE
      A[Index+Gap] = ToSortEl // insert element to sort in its place
    END FOR
  END FOR
  SET Gap = floor(Gap/2)
END WHILE
```

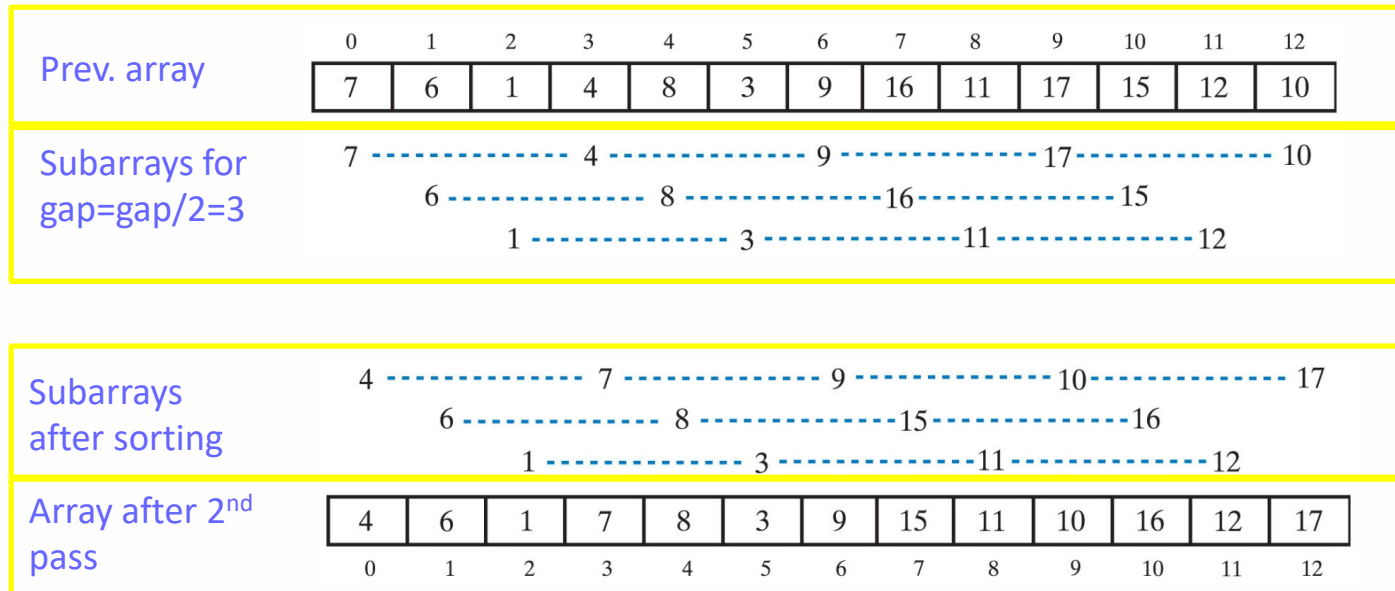
This is an  
Insertion  
Sort on  
subarrays  
separated  
by gap



# Shell Sort in Operation (1)



# Shell Sort in Operation (2)



Last pass: gap = gap/2 = 1.

Loop through array, doing a standard Insertion Sort.

Repeat until have a pass where no swaps are made.



# Shell Sort: Implementation



See **ShellSort.java**

based on code from Carrano & Savitch,  
“Data Structures and Abstractions with Java”

shellSort method

Takes array of Comparable objects  
(i.e. objects that implement Comparable interface)  
Uses a supporting method **insertionSort**

We can also use insertionSort() directly if we wish  
Expect it to be less efficient than shellSort()

# Shell Sort: Complexity

Analysis is complicated: will skip it  
Worst-case complexity:  $O(n^2)$

However, this is because gap is sometimes even  
End up with subarrays that include all elements of an array that was already sorted  
E.g. compare first subarrays when gap=6 and gap=3

To avoid this, round gap up to nearest odd number  
End up with worst-case complexity  $O(n^{1.5})$   
This is done in the implementation on Blackboard

Other gap sequences can improve performance a little more;  
beyond the scope of this topic



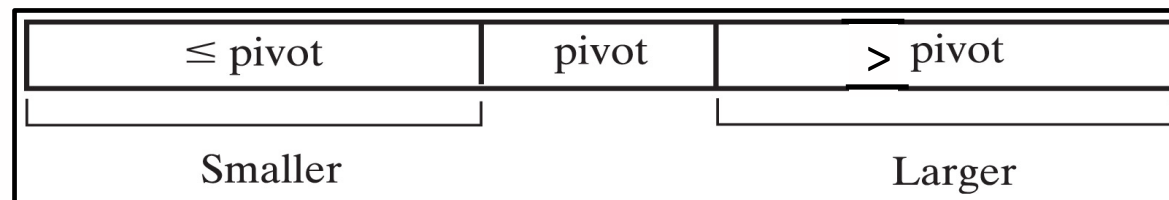
# QuickSort

QuickSort is a **divide-and-conquer** algorithm  
It partitions the array into two sub-arrays that are **partially sorted**:

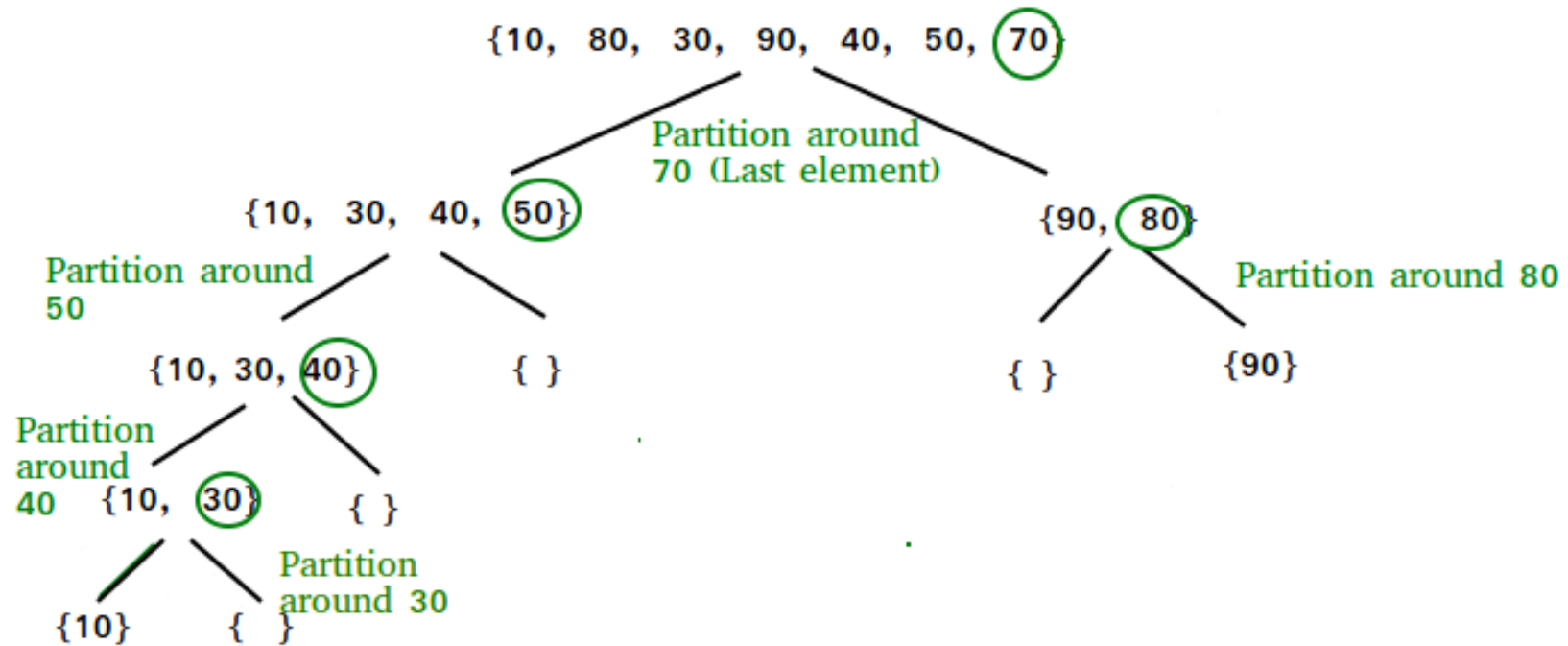
It picks a pivot value, and re-arranged elements so that all elements less than or equal to the pivot value are on the left, and all greater than it are on the right  
Array is now divided into two sub-arrays and pivot value

It then repeats the procedure **recursively** for each sub-array, to further sort each of them

When it has reached the level of a sub-array with just 1 element, that sub-array is sorted  
All sub-arrays are sorted relative to each other, so the whole array is sorted when all sub-arrays are.



# QuickSort: Simple Example



<https://www.geeksforgeeks.org/quick-sort/>



# QuickSort Algorithm

**QuickSort(Array, Left, Right):**

// QuickSorts the subarray from index Left to Right in place

// If sorting full array: Left=0, Right=length-1

If (Left >= Right) Then

return // only one or no value in subarray... so sorted!

Else

1. **Partition** the elements in the subarray Left..Right, so that the pivot value is in place (in position PivotIndex)

-- see next slide

2. **QuickSort** the first subarray Left..PivotIndex-1

3. **QuickSort** the second subarray PivotIndex+1..Right

End If

**Note: this is a recursive algorithm.**

Unsorted Array





# QuickSort: Partitioning the Array (1)

## Basic Idea:

Search from both ends, comparing elements with the pivot value

Find two elements that are out of place:

A big one on the low side of the pivot point

A small one on the high side of the pivot point

Swap these

Keep going until the searches meet in the middle

## Note:

Don't know in advance where pivot index will end up

It's the point where the searches meet in the middle



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# QuickSort: Partitioning the Array (2)

1. Set PivotValue = value of Array[Right]  
(Alternative: mid value of Array[Left],[Middle],[Right])
2. Initialize Up = Left and Down = Right-1
3. Repeat While Up <= Down:
  - While Up <= Down and Array[Up] <= PivotValue  
Increment Up {Scan rightward to find el. larger than pivot}
  - While Down >= Up and Array[Down] >= PivotValue  
Decrement Down {Scan leftward to find el. smaller than pivot}
  - if Up < Down, exchange Array[Up] and Array[Down]
4. Set PivotIndex = Up
5. Put the PivotValue in place, by exchanging  
Array[Right] and Array[PivotIndex]

Unsorted Array



# QuickSort: Implementation

See `QuickSort.java`

Based on code from Goodrich & Tamassia

Principal method is `quickSort()`

Uses a secondary method `quickSortStep()`:  
this is the recursive one

Also requires a Comparator class

I have one for case-insensitive comparison of strings



# QuickSort: Complexity (1)

## Basis of analysis:

To find a single partition, what is the complexity?

(How many times is each of the  $n$  elements visited?)

Sketch out the recursive calls: note that at each call level, no two calls operate on the same section of list

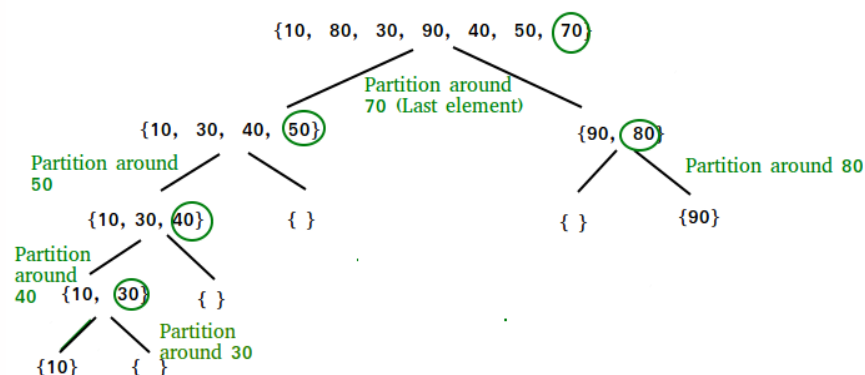
If the overall depth is  $d$ , how does this relate to the number of elements,  $n$ ?

## Result:

Quick sort is  $O(n \log n)$  in the average case

$O(n^2)$  in the worst case:

**What causes worst case? What does sketch look like?**



# QuickSort: Complexity (2)

- Worst case arises when array is already sorted, *and* first element is chosen as the pivot
  - Quite a common occurrence
- Can be easily avoided
  - Pick a pivot at random
- Would get best performance if pivot exactly divides array in 2 (i.e. median value in array)
  - Would make worst case equal to average case
  - Can't achieve this without sorting the array first!
  - Can approximate it:  
mid value of Array[Left],[Middle],[Right])

Note: ideal pivot being close to **median** value is nothing to do with it being near central index of *random* array!



# Sorting Without Comparisons (1)

It can be proven that a sorting algorithm based on *comparisons* and *swapping* is at best  $O(n \log n)$

To improve on this, need completely different approach

Suppose you want to sort items where you know:

Each key is an int in the range 0-20

Each key is different

How could you sort the keys?

18	14	12	10	9	4	19	6	17	7	8	3
----	----	----	----	---	---	----	---	----	---	---	---



# Sorting Without Comparisons (2)

A simple approach:

Create a new array: `NewArray[0..19]`

Make a single pass through the array of items, and put each one in `NewArray` at the position given by its key

Make a single pass through `NewArray`, to copy each item back into the original array, skipping any blanks

Complexity Analysis:

(Will do in class)



# Radix Sort

- Related to the simple approach just described
- Does not compare objects
- Treats array elements as if they were strings of the same length
- Groups elements by a specified digit or character of the string
  - Elements placed into "buckets" which match the digit (or character)





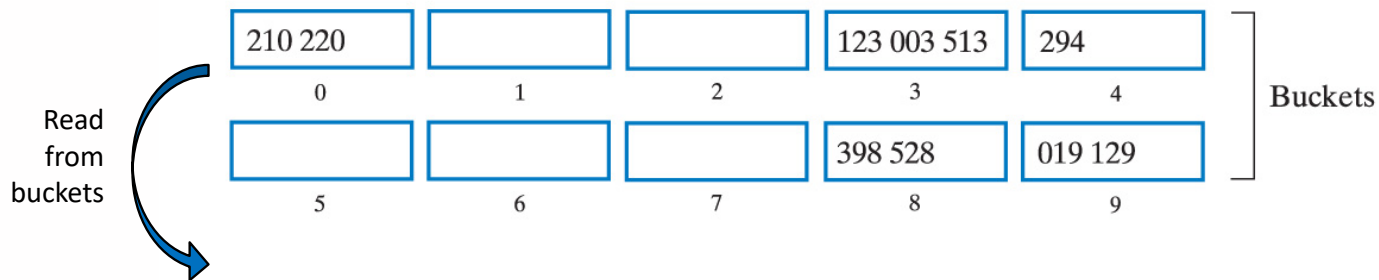
# Radix Sort: Example

(a) 

123	398	210	019	528	003	513	129	220	294
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

 Unsorted array

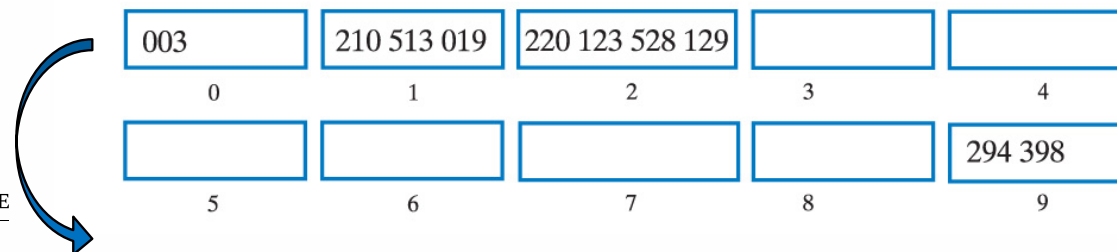
Distribute integers into buckets according to the rightmost digit



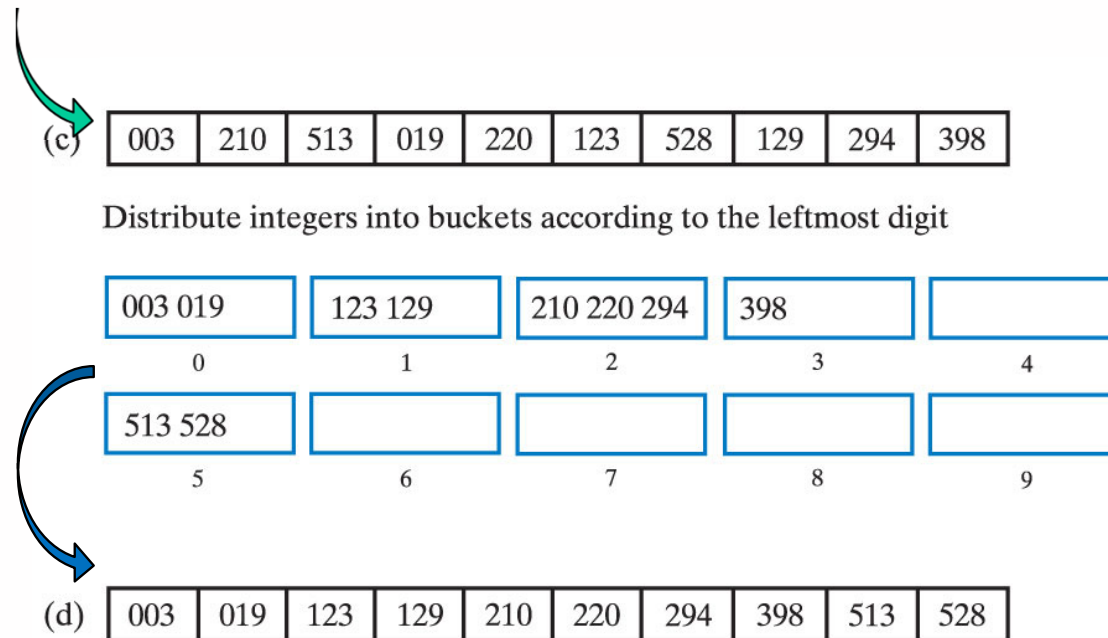
(b) 

210	220	123	003	513	294	398	528	019	129
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Distribute integers into buckets according to the middle digit



# Radix Sort: Example cont'd



# Radix Sort Algorithm

```
Algorithm radixSort (a, first, last, maxDigits)
// Sorts array of positive decimal integers a[first..last], ascending.
// maxDigits is the number of digits in the longest integer.
for (i = 0 to maxDigits - 1)
{
  Clear bucket[0], bucket[1], . . . , bucket[9]
  for (index = first to last)
  {
    digit = digit i of a[index]
    Place a[index] at end of bucket[digit]
  }
  Put contents of bucket[0], bucket[1], . . . , bucket [9] into array a
}
```

## Note on buckets

- Each bucket must be able to store multiple items
- Must retain order in which items were inserted
- Could use an array of linked lists



# Radix Sort: Complexity

If array has  $n$  integers, inner loop repeats  $n$  times

If each integer has max of  $d$  digits, outer loop repeats  $d$  times

Therefore, Radix Sort is  $O(n * d)$

However, if  $d$  is fixed and is much smaller than  $n$ , it is just a constant

A 32-bit integer has fewer than 10 digits

Therefore, Radix Sort is  $O(n)$

Linear complexity!

Much better than  $O(n \log n)$  of other algorithms

Not appropriate for all data, but good when usable

How about sorting text, e.g. surnames?



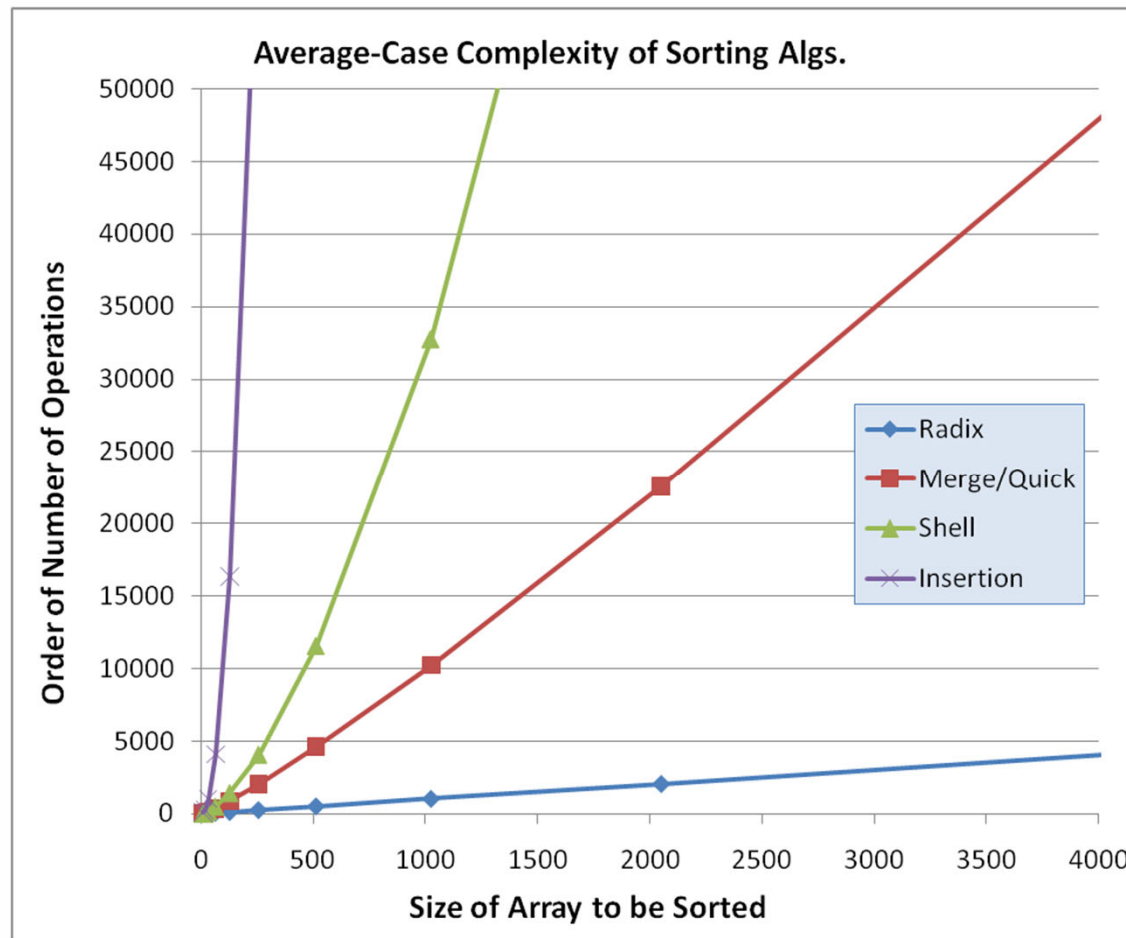
## Summary of Complexity of Sorting Algorithms

	Average Case	Best Case	Worst Case
Radix sort	$O(n)$	$O(n)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Shell sort	$O(n^{1.5})$	$O(n)$	$O(n^2)$ or $O(n^{1.5})$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Reminder: even when two algorithms have same complexity, one can be faster than another:  $O$  analysis ignores constants



# Summary of Complexity of Sorting Algorithms



# Some Sorting Visualisations

The screenshot shows the YouTube channel page for 'AlgoRythmics'. The channel name is at the top, followed by navigation tabs for Videos, Playlists, Channels, Discussion, and About. The 'Videos' tab is selected. Below the navigation, there are filters for 'Date added (newest - oldest)' and 'Grid'. The main content area displays six video thumbnails, each representing a different sorting algorithm visualized with a folk dance. Each thumbnail includes a video player preview, a duration, a title, and view information.

Video Title	Duration	Views	Age
Quick-sort with Hungarian (Küküllőmenti legényes)	6:55	799,930	3 years ago
Merge-sort with Transylvanian-saxon	4:17	237,164	3 years ago
Shell-sort with Hungarian (Székely) folk dance	4:31	329,113	3 years ago
Select-sort with Gypsy folk dance	7:07	295,561	3 years ago
Bubble-sort with Hungarian ("Csángó") folk dance	5:16	844,697	3 years ago
Insert-sort with Romanian folk dance	4:04	336,707	3 years ago



<https://www.youtube.com/user/AlgoRythmics>

Email your suggestions of other good visualisations

School of Computer Science

# What You Achieved in This Topic

- Describe, implement and analyse the complexity of sequential and binary search
- Describe the general operation and algorithm details of a variety of sorting algorithms
- Implement and use these algorithms
- Analyse the algorithmic complexity of the algorithms
- Discuss the relative merits of different sorting algorithms and select the most appropriate for a task.
- **Final note:** see [Blackboard](#) for links to visualisations that you might find helpful. Contact me if you find other good ones!







OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, IT Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

# What You'll Achieve in This Topic

- Define: Node; Arc; Graph; Directed Graph; Directed Acyclic Graph; and terms commonly used with trees
- Explain what tree structures are and provide examples of where they can be used
- Provide formal definitions of general and binary tree
- Write interfaces for Tree, BinaryTree and BinaryNode
- Provide algorithms for traversing a tree in pre-order, post-order, level-order and in-order
- Explain how binary trees may be implemented as linked structures, including the use of Generics

*Additional Reading: Carrano & Savitch, Ch. 24 & 25*



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Terminology

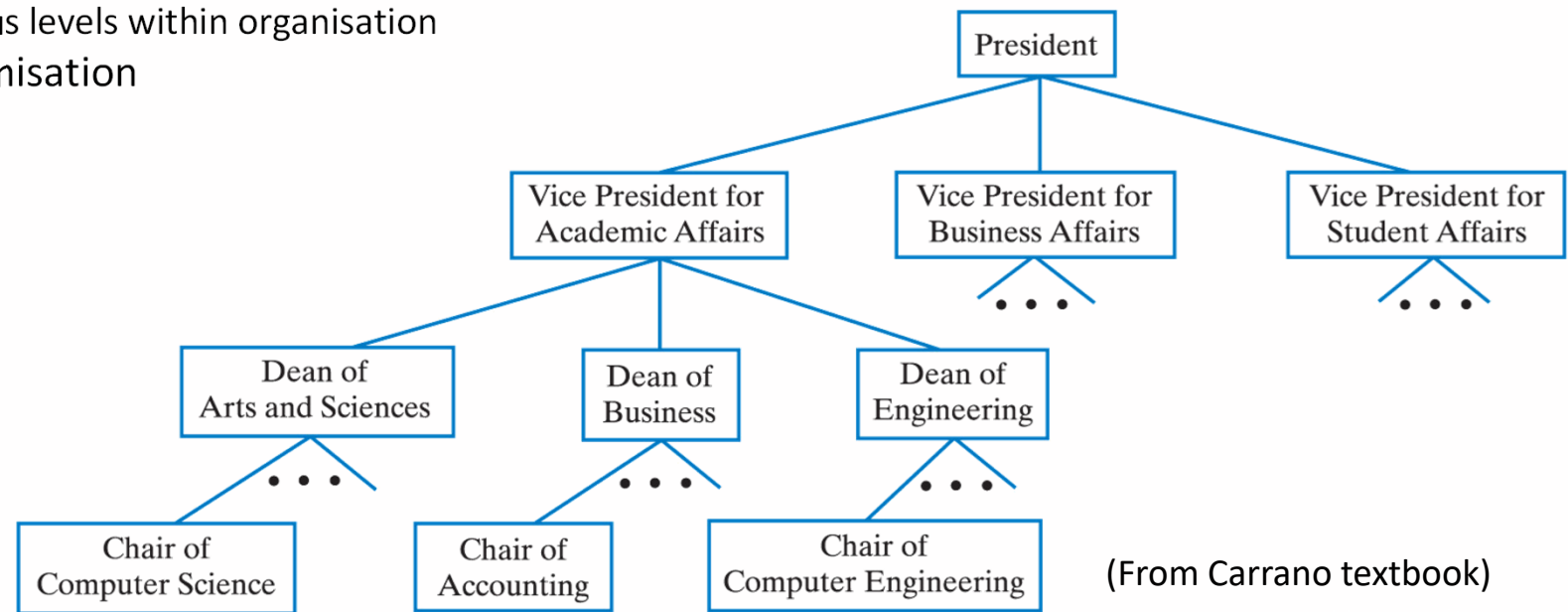
- Nodes & Arcs
- Graphs
- Directed Graphs
- Directed Acyclic Graphs
- Trees
- Binary Trees & Linked Lists

In class:  
Define all of these  
Discuss relationships between them



# Trees: Overview (1)

Previous data structures place data in **linear** order  
But often need to organise data into groups & subgroups  
This is hierarchical classification  
Data items appear at various levels within organisation  
Example: University's organisation



(From Carrano textbook)



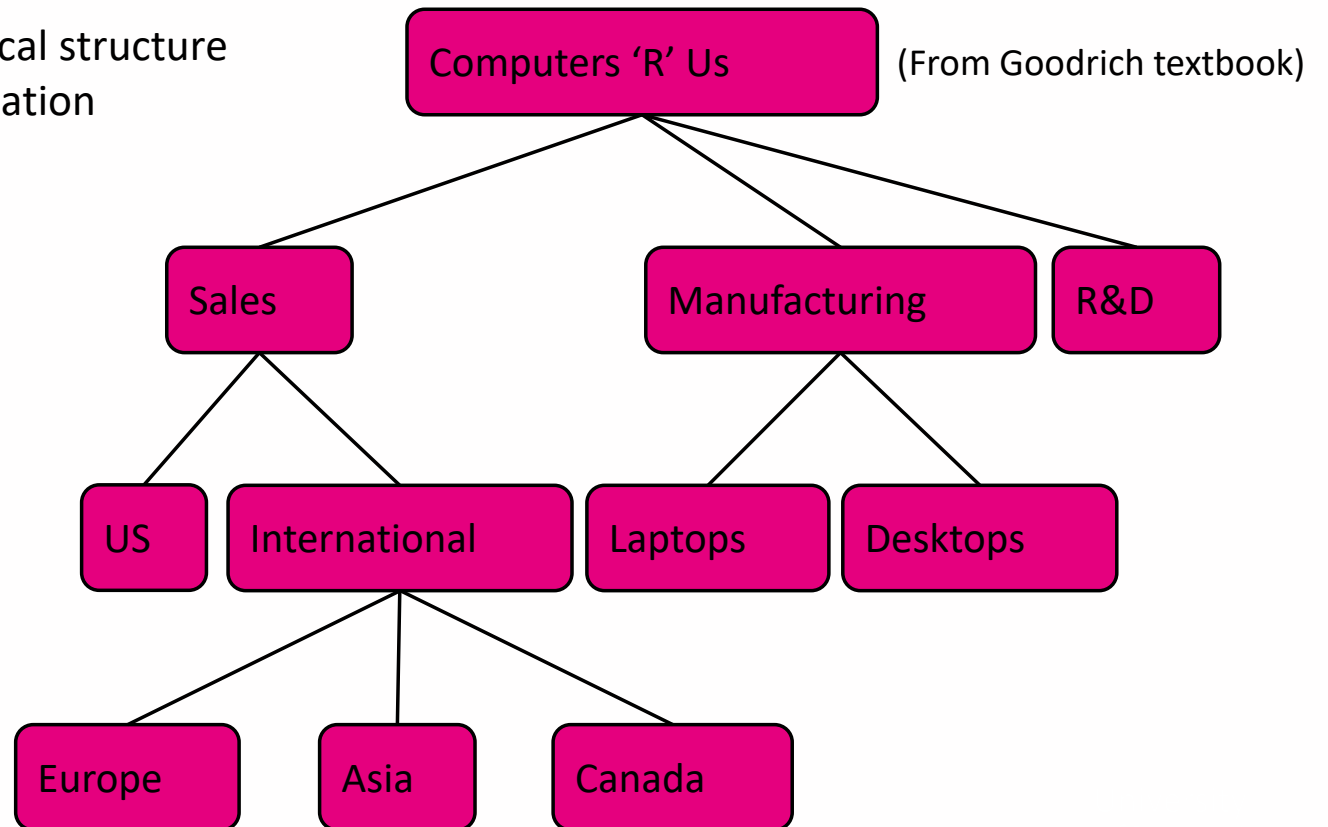
OLLSCOIL NA GAILLIMHĒ  
UNIVERSITY OF GALWAY

# Trees: Overview (2)

In CS, tree is abstract model of hierarchical structure  
Consists of nodes with a parent-child relation

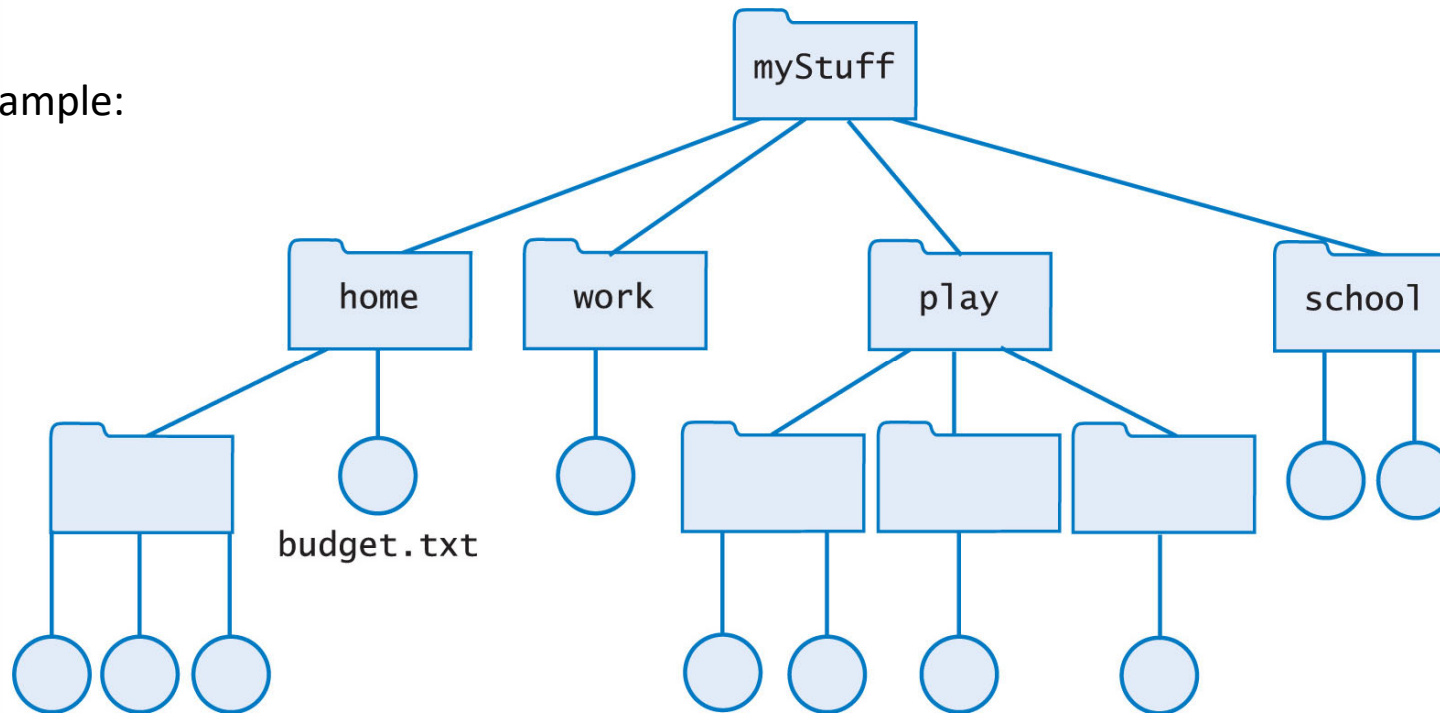
Applications:

- Organization charts
  - File systems
  - Programming environments
- Many others ...



# Trees: Overview (3)

File system example:



(From Carrano textbook)



# Tree Terminology (1)

A tree is:

- A set of nodes
- Connected by edges

The edges indicate relationships among nodes

Nodes arranged in **levels**

Indicate the nodes' hierarchy

Top level is a single node called the **root**

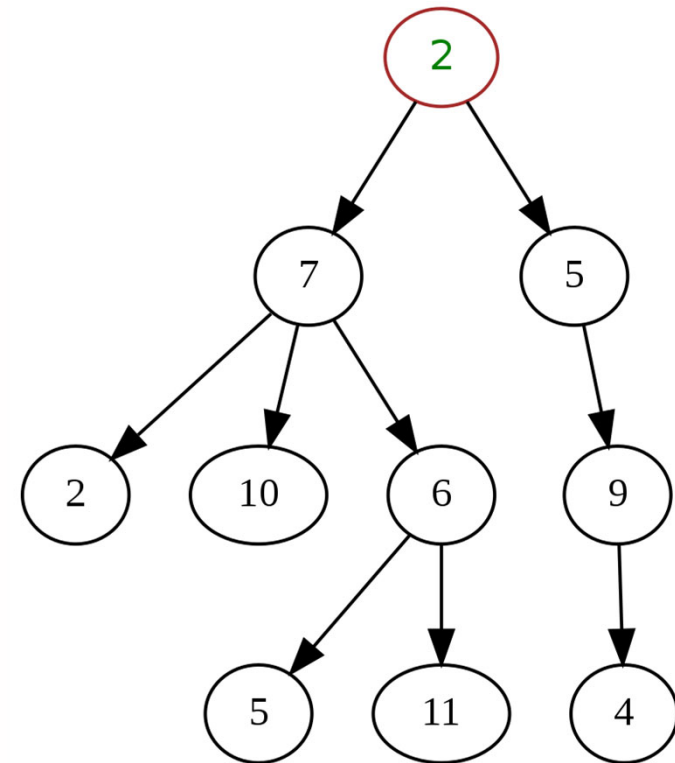
Nodes at a given level are **children** of nodes of previous level

Node with children is the **parent** node of them

Nodes with same parent are **siblings**

The only node with no parent is the root node

All others have one parent each



# Tree Terminology (2)

Node with no children is a **leaf node** or **external node**

All other nodes are **internal nodes**

A node is reached from the root by a **path**

**Length** of path is the number of edges that compose it

This is also called **depth of the node**

**Height** of a tree is the number of levels in the tree

The number of nodes along the longest path

= maximum depth + 1

Note we talk of **depth** of a node but **height** of the tree overall

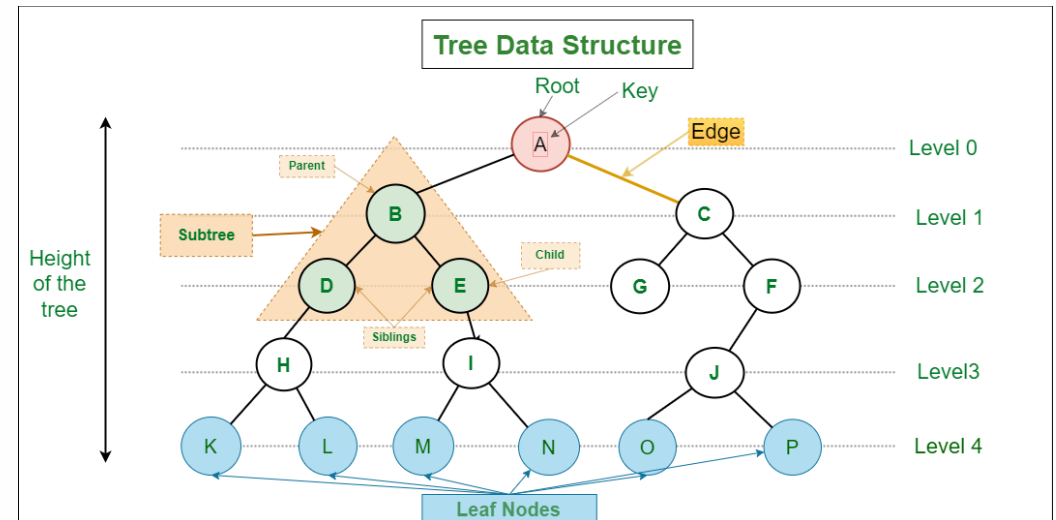
**Ancestors** of a node:

Parent, grandparent, great-grandparent, etc

**Descendants** of a node: child, grandchild, etc

**Subtree** of a node is a tree that has that node as its root

Node plus all its descendants and all arcs connecting them



<https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/>



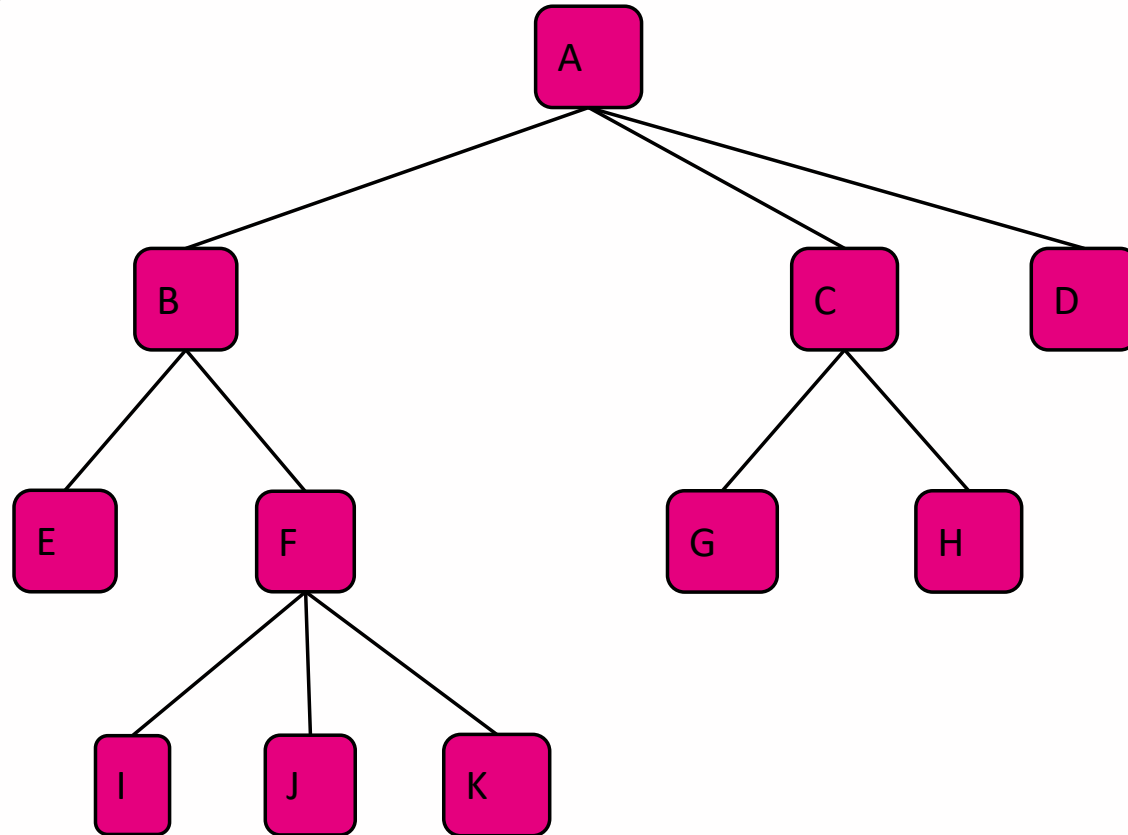
OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY



# Tree Terminology (3)

For this tree, what are ...

- The root?
- The leaves?
- The internal nodes?
- The parent of F?
- The ancestors of F?
- The children of B?
- The descendants of B?
- The siblings of D?
- The height of the tree?
- The depth of G?
- The subtree of C?



# Binary Trees (1)

Binary tree: a tree with the following properties:

Each internal node has at most two children (**exactly** two if it's a “**proper**” binary tree)

The children of a node are an **ordered pair**

Call the children of an internal node

**Left** child and **Right** child

Alternative definition: binary tree is either:

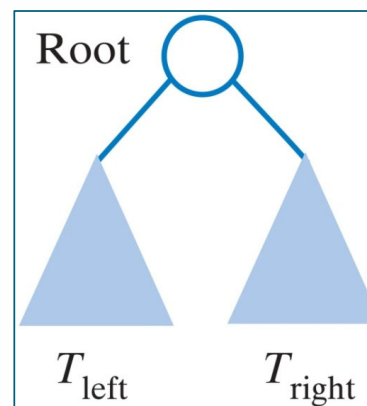
Just a single node, or

a tree whose root has an ordered pair of children, each of which is a binary tree

That's a **recursive** definition:

will use recursive functions and data structures

**Note: Non-binary trees are termed General Trees**



# Binary Trees (2)

**Full** binary tree (only achievable for some numbers of nodes):

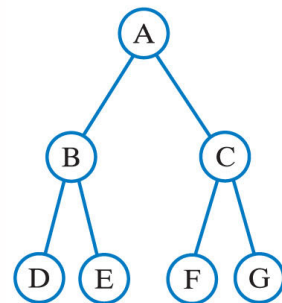
Every internal node has exactly two children (proper binary tree) AND all leaves are at the same level

**Complete** binary tree (achievable for any number of nodes):

Full to second-last level

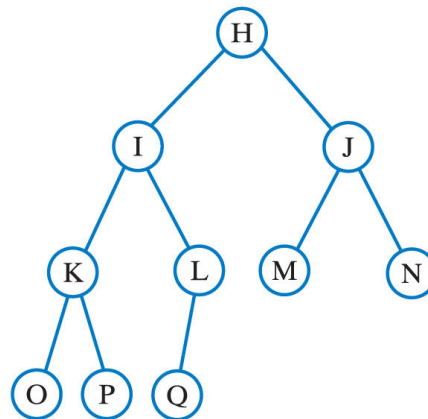
AND leaves on last level filled from left to right

(a) Full tree



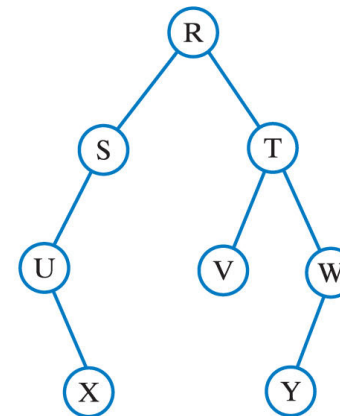
Left children: B, D, F  
Right children: C, E, G

(b) Complete tree



(Carrano textbook)

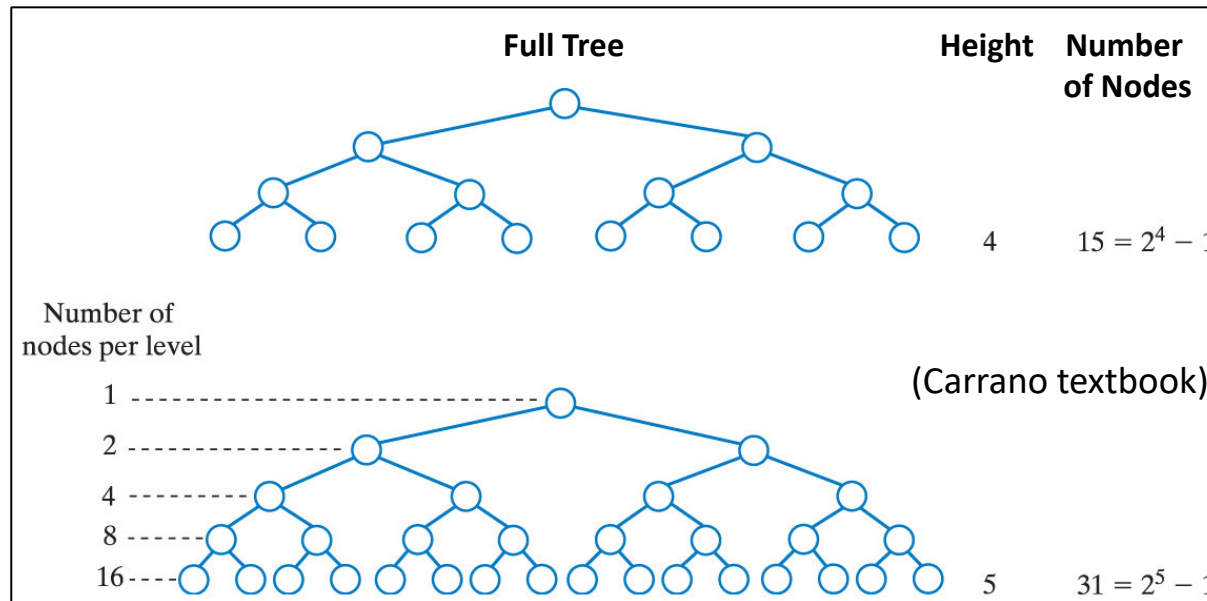
(c) Tree that is not full and not complete



# Binary Trees (3)

The height of a **complete** or **full** binary tree with  $n$  nodes is  $\log_2(n + 1)$

Is this an OK approximation of the height of any BT?



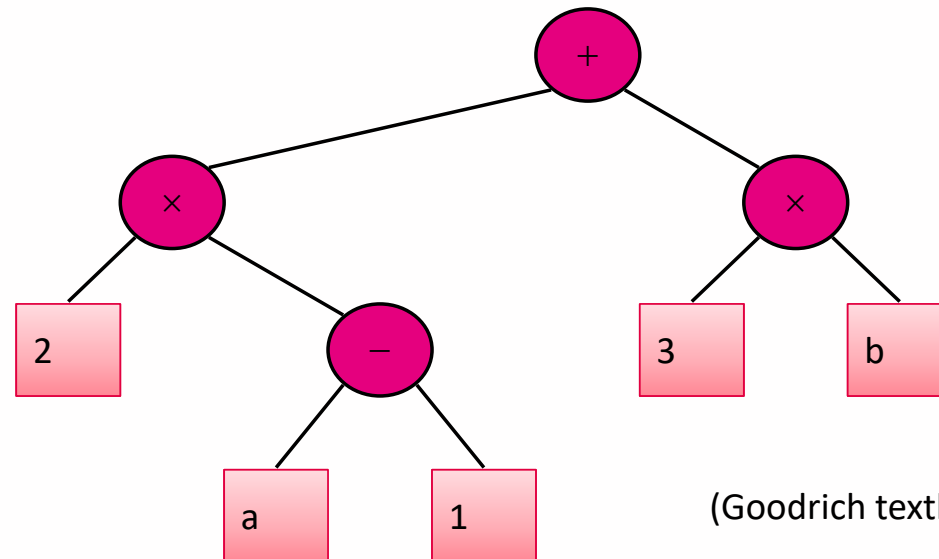
# Example: Arithmetic Expression Tree

Binary tree associated with an arithmetic expression

Internal nodes: operators

Leaves: operands

Example: arithmetic expression tree for the expression  $(2 \times (a - 1)) + (3 \times b)$



(Goodrich textbook)



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

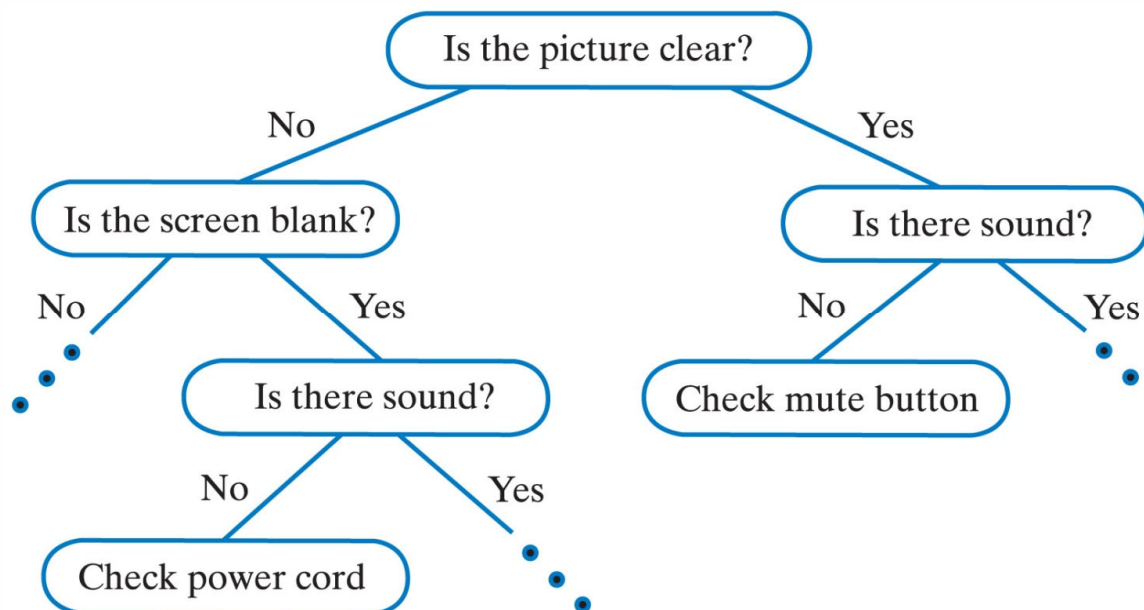
# Example: Decision Tree

Binary tree associated with a decision process

Internal nodes: questions with yes/no answer

External nodes: decisions

Example: Trouble-shooting TV



(Carrano textbook)



# Binary Tree: Linked Structure Representation

A node is represented by an object storing:

Element

Left child node

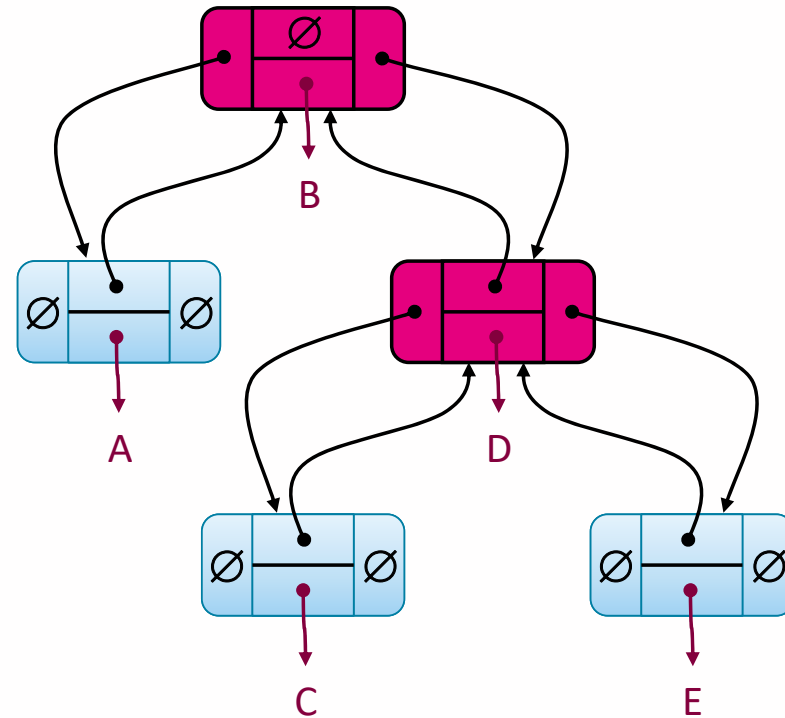
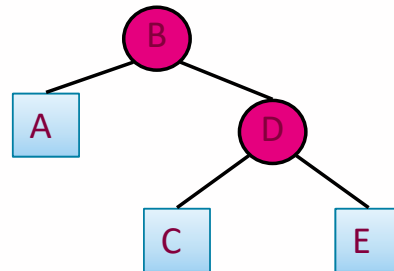
Right child node

Parent node

["optional": not always done]

The Binary Tree

just stores a root node



# Binary Tree Implementation [1]

Taken with permission from Carrano & Savitch

Uses **generics**: see next slides

Define with a generic name, e.g. 'T', to represent a datatype, which is replaced later by an actual type, eg String or Double

Interfaces (see next slides)

[TreeInterface.java](#): tree interface

[BinaryTreeInterface.java](#): extends TreeInterface

[BinaryNodeInterface](#): For storing data, left node, right node



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY



# Binary Tree Implementation [2]

## Classes Implementing Interfaces

[BinaryTree.java](#): implements BinaryTreeInterface

[BinaryNode.java](#): implements BinaryNodeInterface

## Test class (will look at this first)

[BinaryTreeDemo](#): class with main( ),  
demonstrating how to use the code



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

All on Blackboard

School of Computer Science

Time for a short diversion from our discussion of trees to learn what Generics are in OOP languages such as Java ....



# About Generics (1)

The `< >` operators relate to concept of Generics:

Used to specify a specific type parameter for a generic collection class

Avoids having to cast objects in **add**, **set**, **remove**

Big advantage: type checking at compile time

Other OOP languages have similar concept, e.g. templates in C++

Without generics, compile-time type checking is impossible, since we don't have a type specification for the list.



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# About Generics (2): Example

ArrayList: Part of Java Collections Framework

Standard library of pre-built data structures

Underlying storage is an array; ArrayList class looks after resizing it as required

Can be used with or without Generics notation

ArrayList code without generics:

```
ArrayList words = new ArrayList(); // Holds Objects
words.add("hello");
String a = (String)words.get(0); // note cast
```

ArrayList parameterised to specifically hold Strings:

```
ArrayList<String> words = new ArrayList<String>();
words.add("hello");
String a = words.get(0); // no cast needed
```



# About Generics (3)

Creating a Generic Collection:

Example: code from List & Iterator interface definitions:

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}  
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

Here, **E** is a generic placeholder for a datatype  
List references in code will substitute a valid  
class name for E, e.g.

```
List<String> words = new ArrayList<String>();
```



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

# Back to Tree Structures!



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Tree Interface

In this design, most of the work is done at the node level: see **BinaryNode** interface (to follow)

Uses generics to specify node type: T is a Node class

```
public interface TreeInterface < T > {  
    public T getRootData();  
    public int getHeight();  
    public int getNumberOfNodes();  
    public boolean isEmpty();  
    public void clear();  
}
```

**TreeInterface.java**



## BinaryTree Interface: extends TreeInterface

Just one extra method, `setTree( )`

Supply data for the root node,  
References to the left and right subtrees  
Subtrees are themselves binary trees

```
public void setTree(T rootData,  
    BinaryTreeInterface<T> leftTree,  
    BinaryTreeInterface<T> rightTree);
```

BinaryTreeInterface.java



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY



# BinaryNode Interface

```
// get and set data stored at this node
public T getData();
public void setData(T newData);
// get and set left [right] child of the node
public BinaryNodeInterface<T> getLeftChild();
public void setLeftChild(BinaryNodeInterface<T> leftChild);
// test whether node has left [right] child, or is leaf
public boolean hasLeftChild();
public boolean hasRightChild();
public boolean isLeaf();
// get height / no. of nodes in the subtree rooted at this node.
public int getHeight();
public int getNumberOfNodes();
// copy the subtree rooted at this node.
public BinaryNodeInterface<T> copy();
```

BinaryNodeInterface.java



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# Traversing a Tree

Goal: visit all nodes

**Visiting** a node = Processing the data within a node

This is the action performed on each node during traversal of a tree

Note: a traversal can pass through a node without visiting it at that moment

For a binary tree

Visit the root

Visit all nodes in the root's left subtree

Visit all nodes in the root's right subtree

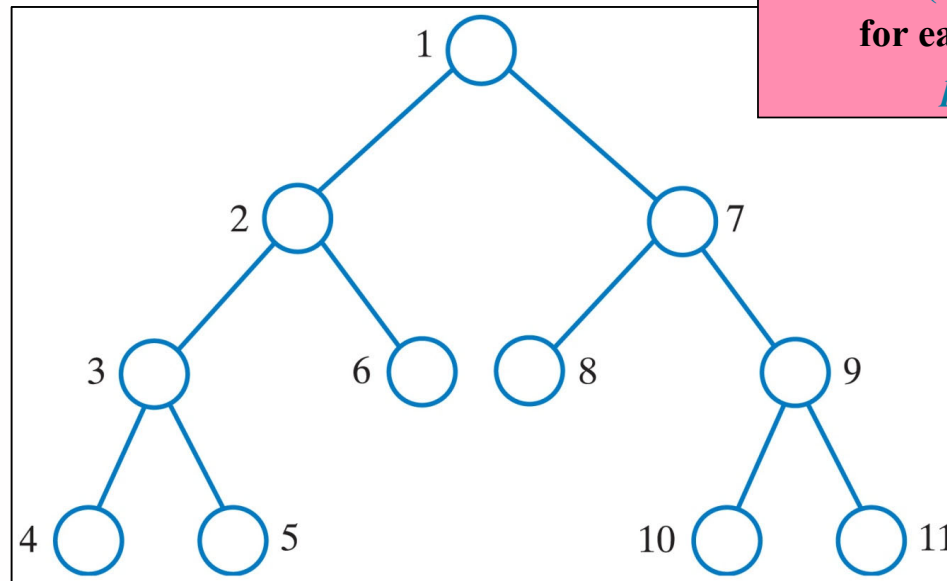
Depending on which order you perform these visits, you get different traversals



# Pre-Order Traversal

For pre-order traversal:  
visit root **before** the subtrees

```
Algorithm preOrder(v)  
  visit(v)  
  for each child w of v  
    preOrder(w)
```

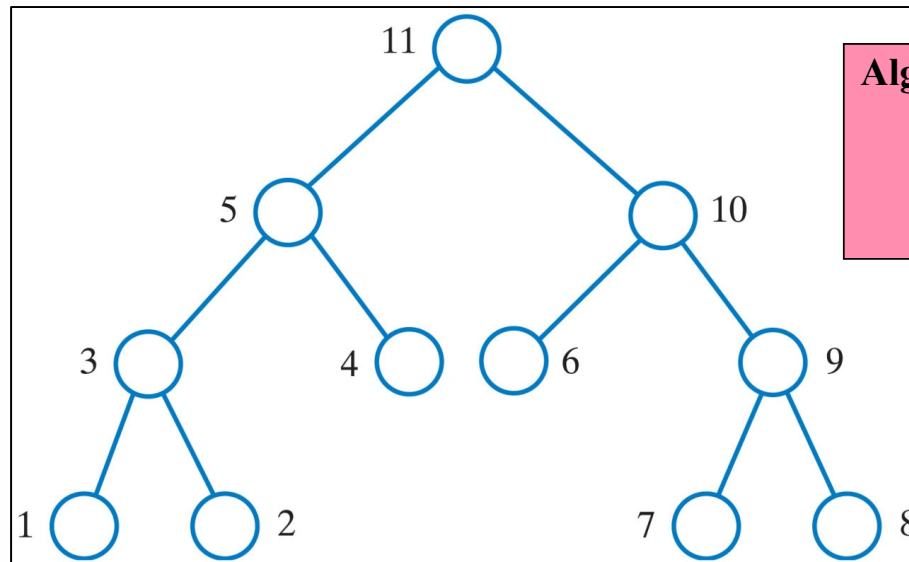


# Post-Order Traversal

For post-order traversal:

visit root **after** visiting the subtrees

This is the classic **depth-first** traversal algorithm,  
though Pre-Order and In-Order are also depth-first



Algorithm *postOrder(v)*  
for each child *w* of *v*  
*postOrder(w)*  
*visit(v)*



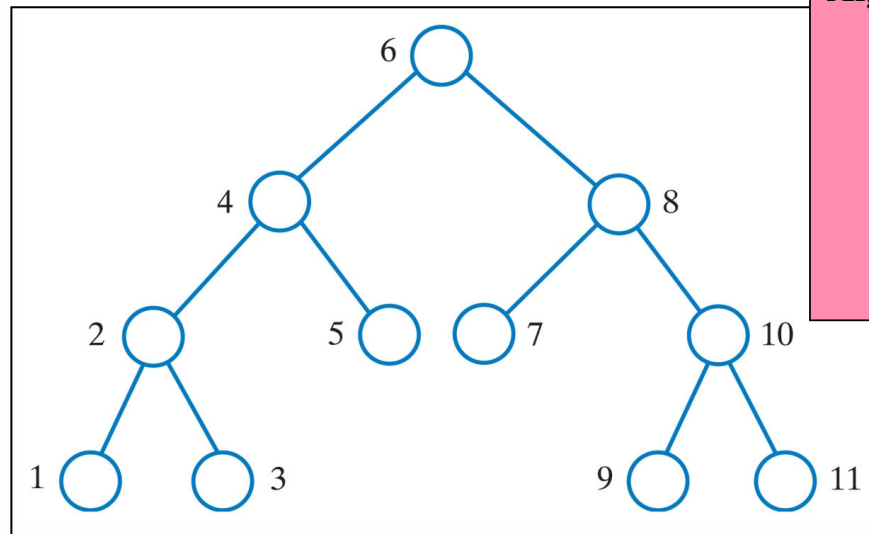
# In-Order Traversal

For in-order traversal:

visit root **between** visiting the subtrees

This is defined only by binary trees

Undefined for general trees



**Algorithm** *inOrder(v)*

**if** *hasLeftChild(v)*

*inOrder(getLeftChild(v))*

*visit(v)*

**if** *hasRightChild(v)*

*inOrder(getRightChild(v))*



# Another Way of Visualising Traversals

In-Order, Pre-Order and Post-Order algorithms all move through tree in essentially the same way

Start at the top, go down left side and around whole tree

You pass by nodes multiple times, unless they have no children

As you pass by a node, whether or not you visit it is determined by Order

Pre-Order: visit node before its children

Post-Order: visit node only after visiting all of its children

In-Order: visit node after left child and before right child

Note: this is just a way of thinking about traversals

Not a substitute for the formal algorithms



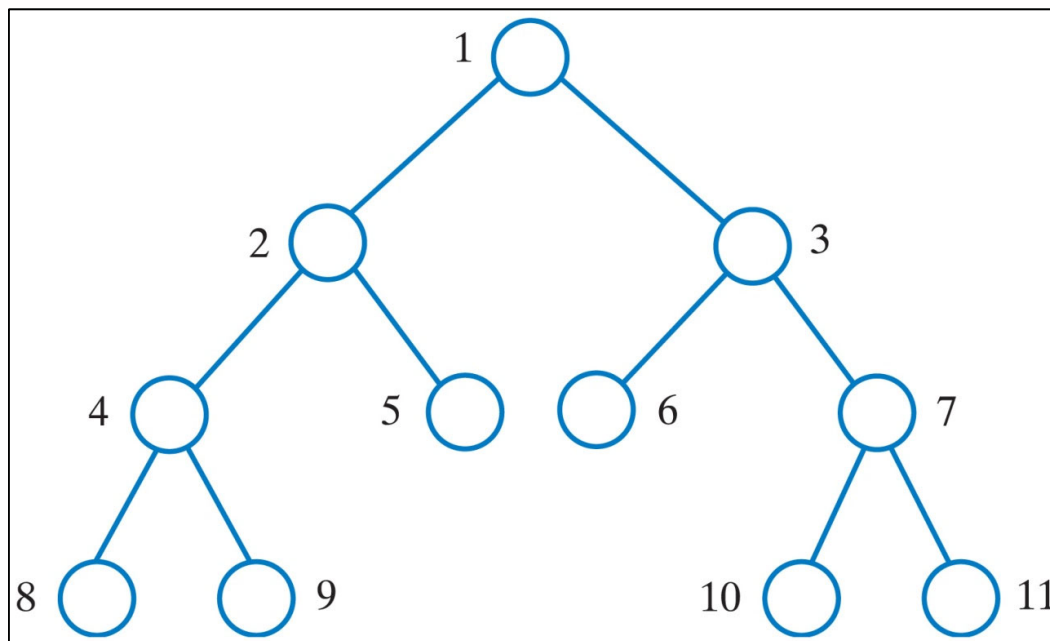
# Breadth-First Traversal

Also known as Level-Order Traversal:

begin at the root, visit nodes one level at a time

How do we do this?

Will discuss it in class



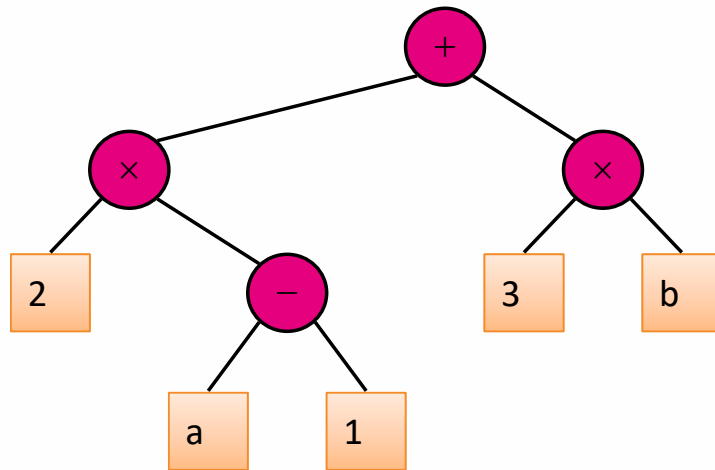
# Example: Print Arithmetic Expression

Specialization of in-order traversal

print operand or operator when visiting node

print "(" before traversing left subtree

print ")" after traversing right subtree



$$((2 \times (a - 1)) + (3 \times b))$$

**Algorithm** *printExpression(v)*

**if** *hasLeftChild(v)*

*print*("(")

*printExpression* (*getLeftChild(v)*)

*print*(*v.getData()*)

**if** *hasRight(v)*

*printExpression*(*getRightChild(v)*)

*print* (")")



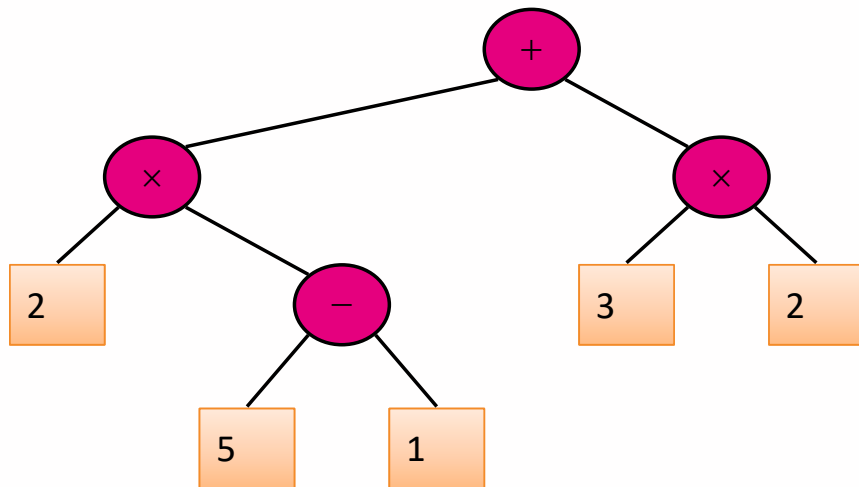


# Example: Evaluate Arithmetic Expression

Specialization of post-order traversal

recursive method returning the value of a subtree

when visiting an internal node, combine the values of the subtrees



**Algorithm** *evalExpr(v)*

**if** *isLeaf(v)*

**return** *v.getData()*

**else**

*x* ← *evalExpr(getLeftChild(v))*

*y* ←

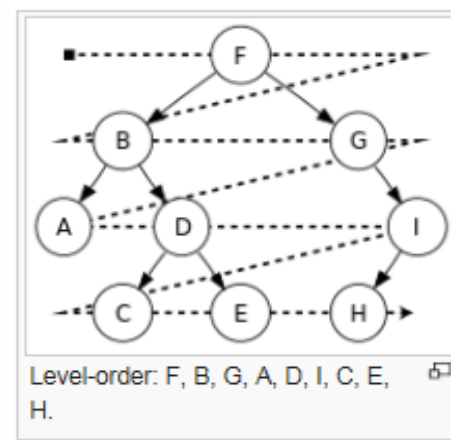
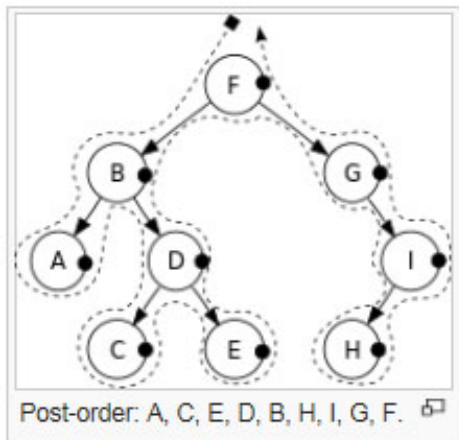
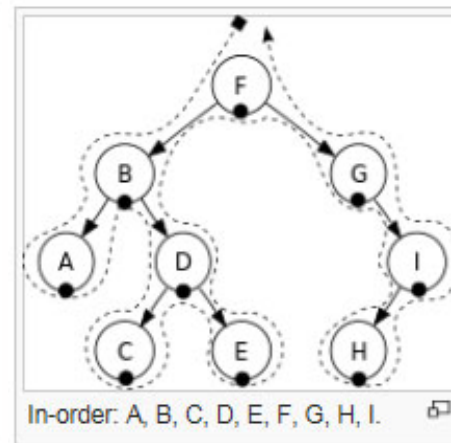
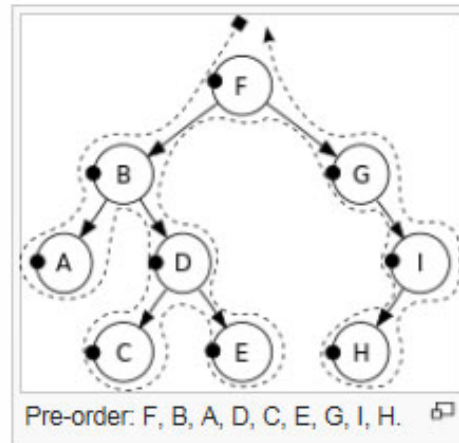
*evalExpr(getRightChild(v))*

◇ ← operator stored at *v*

**return** *x* ◇ *y*



# Tree Traversals Summary [Wikipedia]



# Complexity of Tree Operations

For tree with  $n$  nodes,  $d$  depth

If full/complete binary tree,  $d = \log(n)$

Storage requirement:  $O(n)$

Constant “overhead” per node for storing links

Running times:

get/set Data; get/set Left/Right Child:  $O(1)$

isRoot; isLeaf; hasLeft/RightChild, etc.  $O(1)$

Traversals (including searching for a given value)  $O(n)$

getNumberOfNodes ? (Do in class)

getHeight ? (Do in class)

Could store variables to make the last two  $O(1)$



# What You Achieved in This Topic

- Define: Node; Arc; Graph; Directed Graph; Directed Acyclic Graph; and terms commonly used with trees
- Explain what tree structures are and provide examples of where they can be used
- Provide formal definitions of general and binary tree
- Write interfaces for Tree, BinaryTree and BinaryNode
- Provide algorithms for traversing a tree in pre-order, post-order, level-order and in-order
- Explain how binary trees may be implemented as linked structures, including the use of Generics.





OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, CS Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

# What You'll Achieve in This Topic

- Explain the structure and use of Binary Search Trees, including algorithms to process them
- Analyse complexity of Search Tree algorithms
- Discuss the distinctions between balanced and unbalanced Search Trees, and how balance is achieved
- Describe and use AVL Trees, including algorithms to operate on them
- List and discuss other forms of Search Tree
- Understand how to implement these data structures in Java and demonstrate how to use them



*Additional Reading: Carrano & Savitch, Ch. 26*



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Binary Search Trees

## Search Tree:

Organises its data so that search is efficient

## Binary Search Tree:

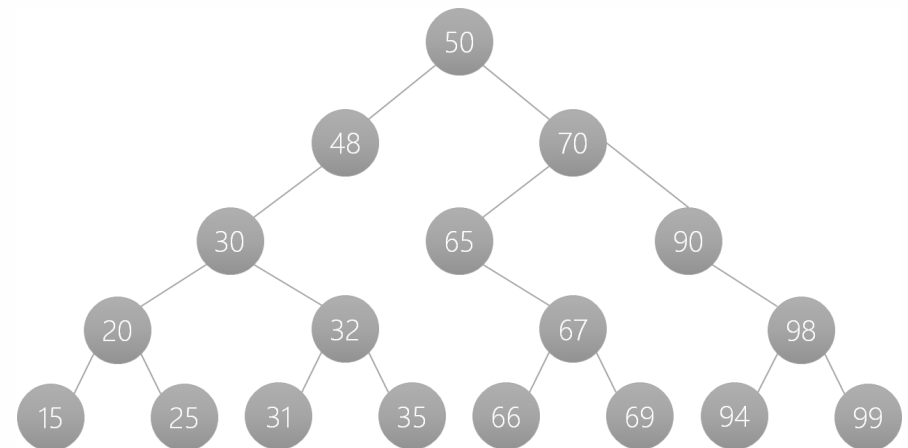
A binary tree

Nodes contain **Comparable** objects

A node's data is **greater than** the data in left subtree

A node's data is **less than** the data in right subtree

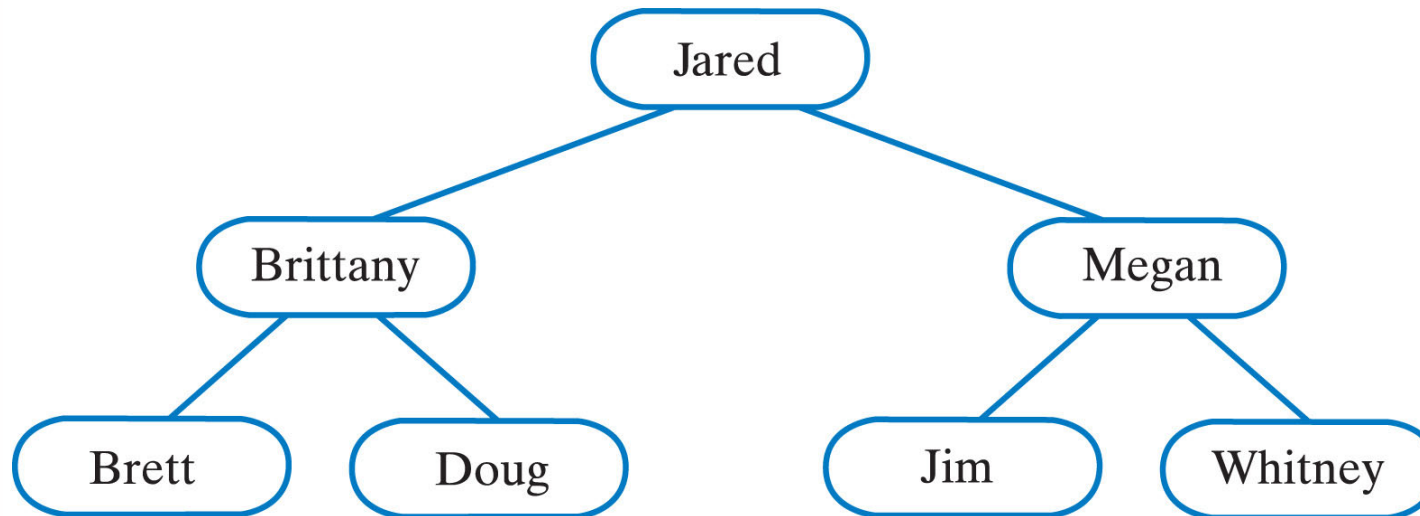
(no duplicates allowed usually)



An **in-order** traversal of BST will visit all nodes in ascending order



## Example: BST of Names



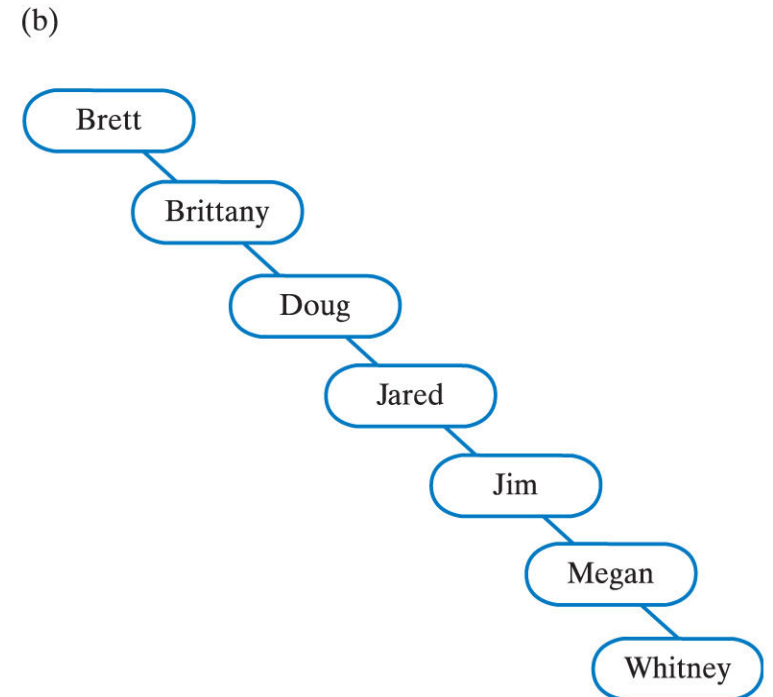
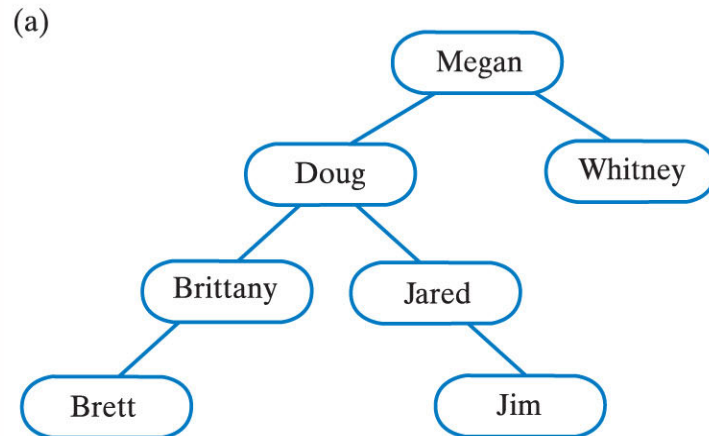
(Diagrams taken from Carrano textbook: reproduced with permission)





# BSTs Are **Not Uniquely** Structured

Depends on what node is *chosen as root*  
and order in which all other nodes are added



# BST Searching Algorithm

```
Algorithm getEntry(rootNode, object)
// Searches a BST (specified by its root node) for a given object.
// Returns the node containing the object, or null if not found.
if (rootNode is empty)
    return null;
else if (object == rootNode.Data)
    return rootNode;
else if (object < rootNode.Data)
    return getEntry (rootNode.leftChild, object) ; // recursive
else
    return getEntry(rootNode.rightChild, object); // recursive
```



# BST Searching Algorithm

Complexity analysis:

Will do in class ...



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Search Tree Interface

```
public interface SearchTreeInterface <T> extends TreeInterface <T> {  
    // Searches for a specific entry in the tree: returns true if found.  
    public boolean contains (T entry);  
    //Retrieves a specific entry in the tree: returns object ref or null  
    public T getEntry (T entry);  
    // Adds a new entry to the tree.  
    // If it matches an object in the tree already, replaces the old object  
    // and returns the old object. Otherwise returns null.  
    public T add (T newEntry);  
    // Removes a specific entry from the tree.  
    public T remove (T entry);  
    // NOT INCLUDED: Creates iterator that traverses all entries in tree.  
    // public Iterator < T > getInorderIterator ();  
}
```



## Reminder from last topic: Tree Interface

In this design, most of the *work* is done at the node level: see **BinaryNode** interface

Uses generics to specify node type: T is a Node class

```
public interface TreeInterface < T > {  
    public T getRootData();  
    public int getHeight();  
    public int getNumberOfNodes();  
    public boolean isEmpty();  
    public void clear();  
}
```



# Note on BST Implementation

Source code: `BinarySearchTreeR.java`

Taken from Carrano textbook (with permission)

Illustrations on following slides also taken from there

Uses the BinaryTree implementation

**R**: Uses Recursive versions of algorithms

Neater than equivalent iterative algorithms

Iterative versions may be more efficient

You should aim to be able to read, understand and use the BST implementation provided

Not necessarily re-write them from scratch

More important to be able to **describe** algorithms.



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

# BinarySearchTree: **Add** Entry

Recursively step through tree

Same as when searching for an entry

If a matching entry is found

Replace it

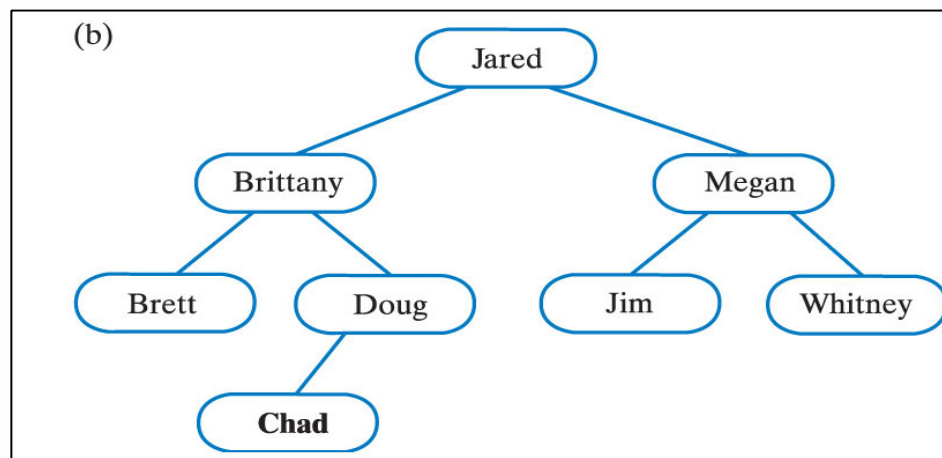
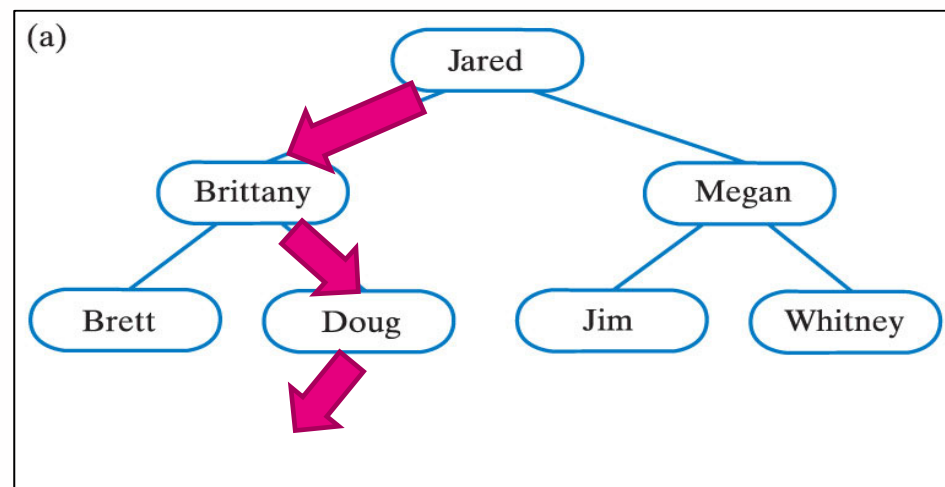
If no match found, will end at node with no child on the relevant side

Add the new node as the child on that side



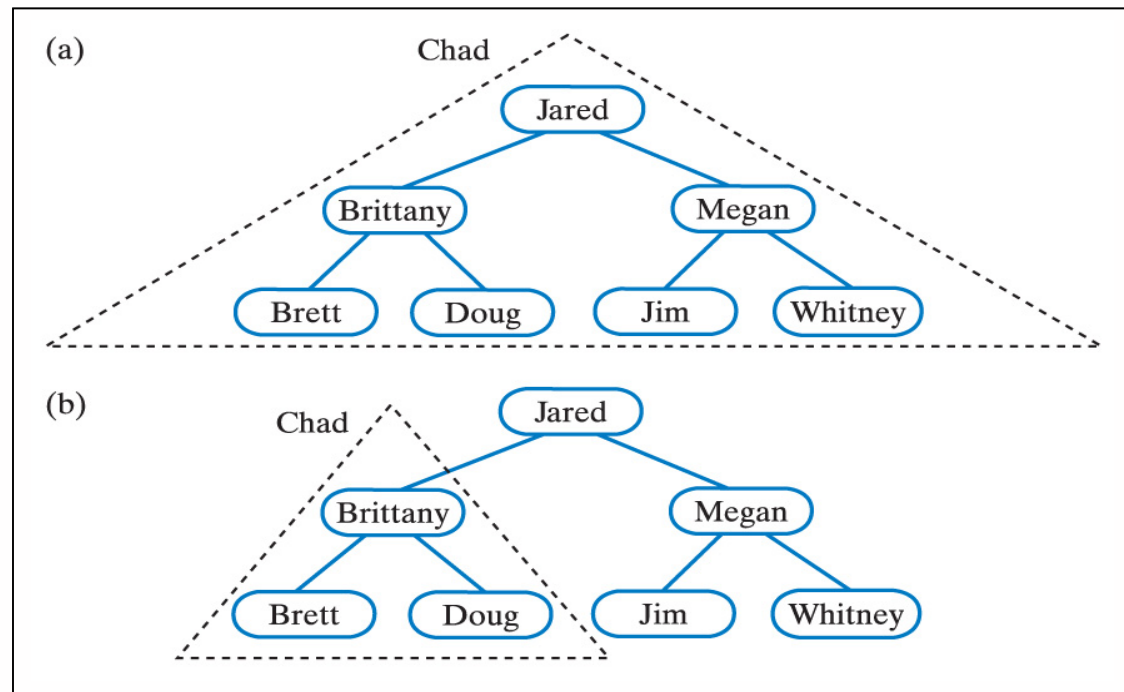
# BinarySearchTree: **Add** Entry

Binary search tree  
before and after  
adding name 'Chad'





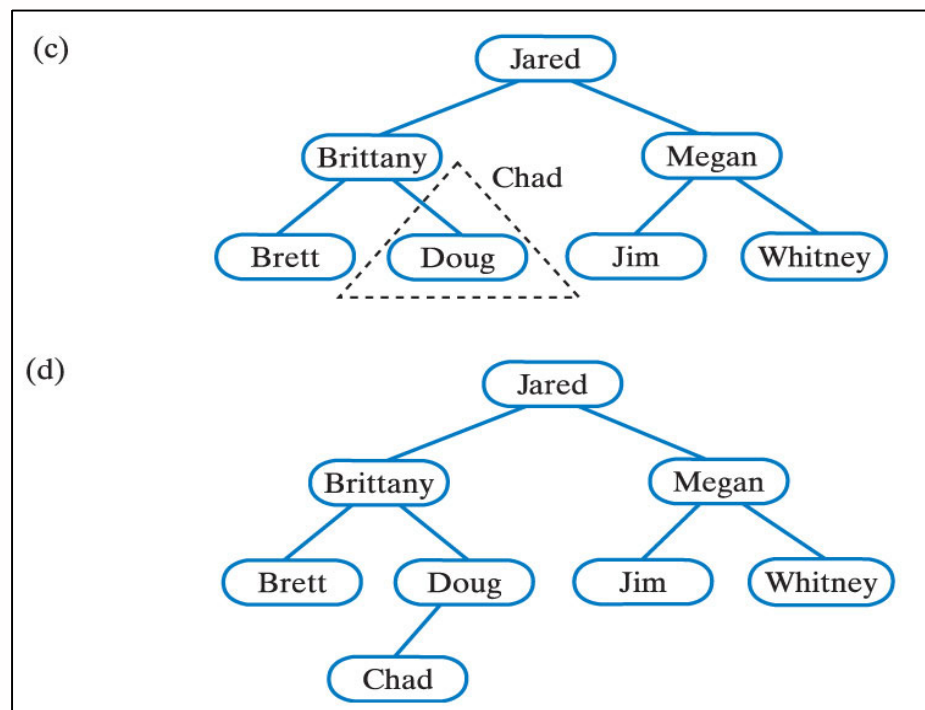
# BinarySearchTree: **Add** Entry



Recursively adding *Chad* to smaller subtrees of BST ...



# BinarySearchTree: Add Entry

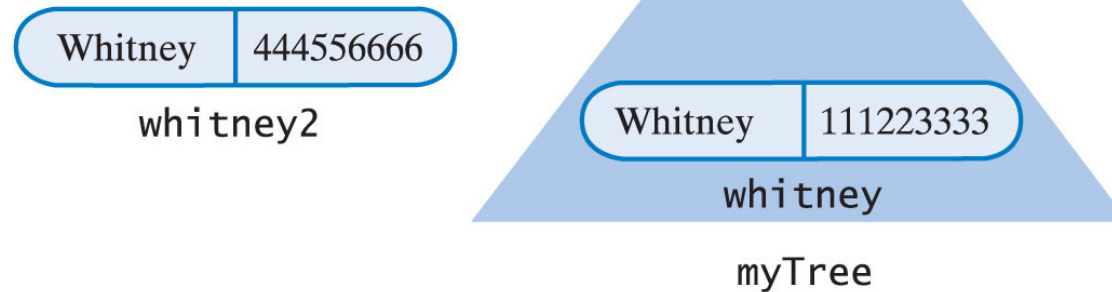


No match found, so end at a node without a left (in this case) child.  
Set this node's left child to be the new node.

# BinarySearchTree: Add Entry (Replace)

- **If object being added matches an object in tree already**  
Replace the old object with the new one  
Return the old object

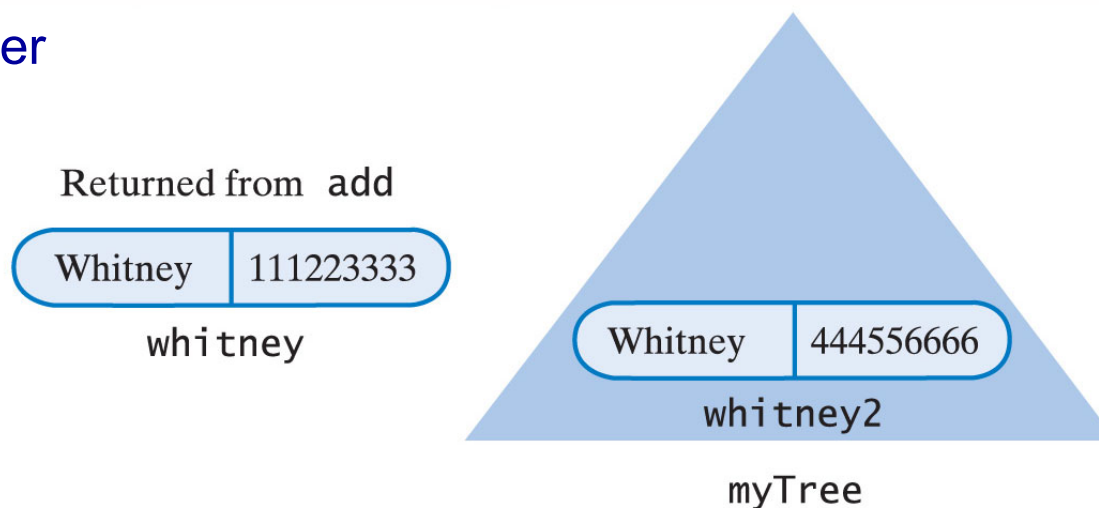
Before ...



# BinarySearchTree: Add Entry (Replace)

- **If object being added matches an object in tree already**  
Replace the old object with the new one  
Return the old object

... After



# BinarySearchTree: **Remove** Entry

The **remove** method must receive an entry to be matched in the tree

If found, it is removed

Otherwise, the method returns null

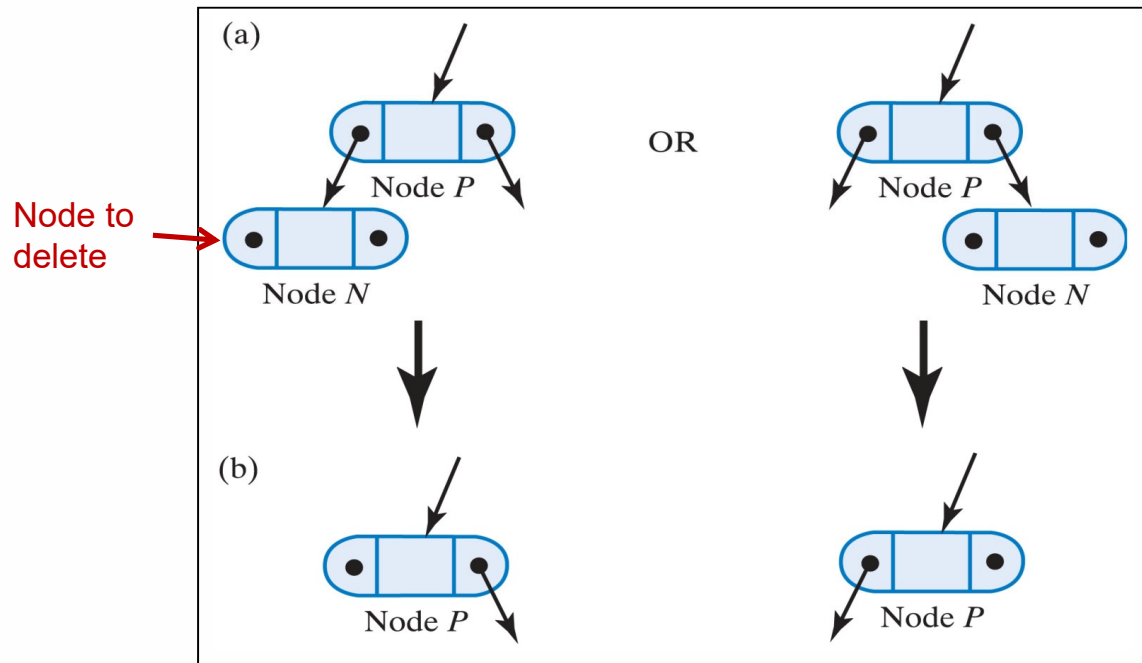
A slightly complex procedure ...

Three cases

1. The node has no children, it is a leaf (simplest case)
2. The node has one child
3. The node has two children



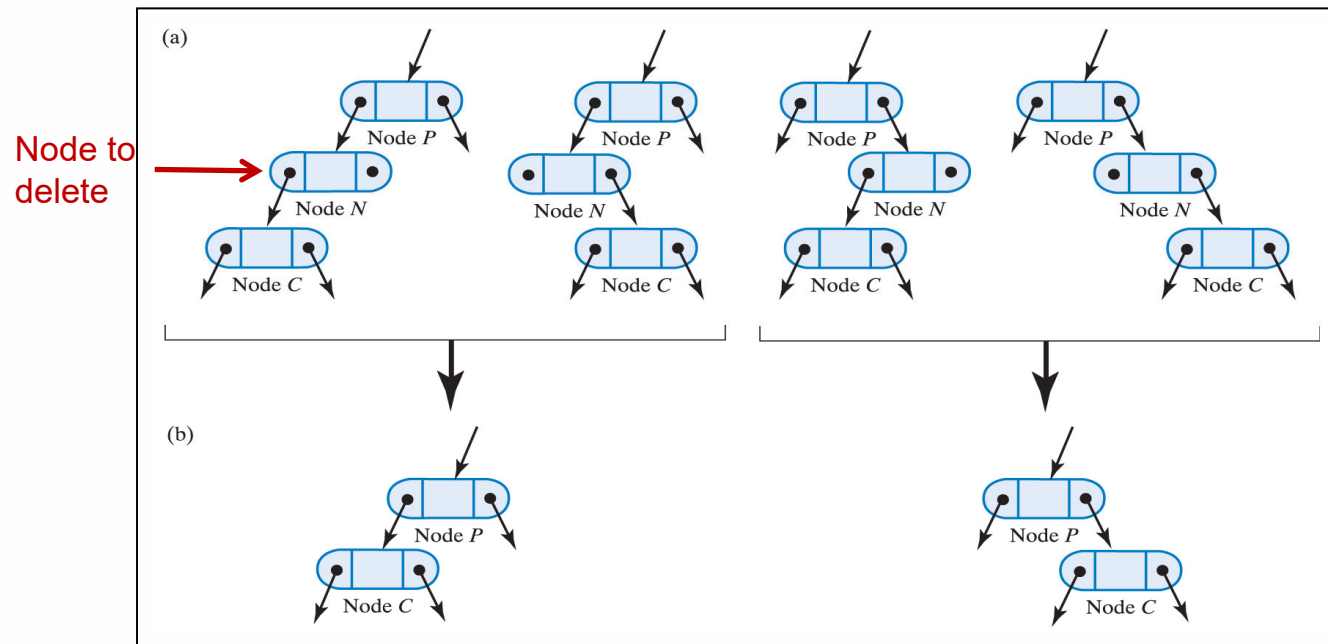
# Remove Entry: Leaf



1. Node is a leaf:  
Just delete node, set parent's pointer to **null**.



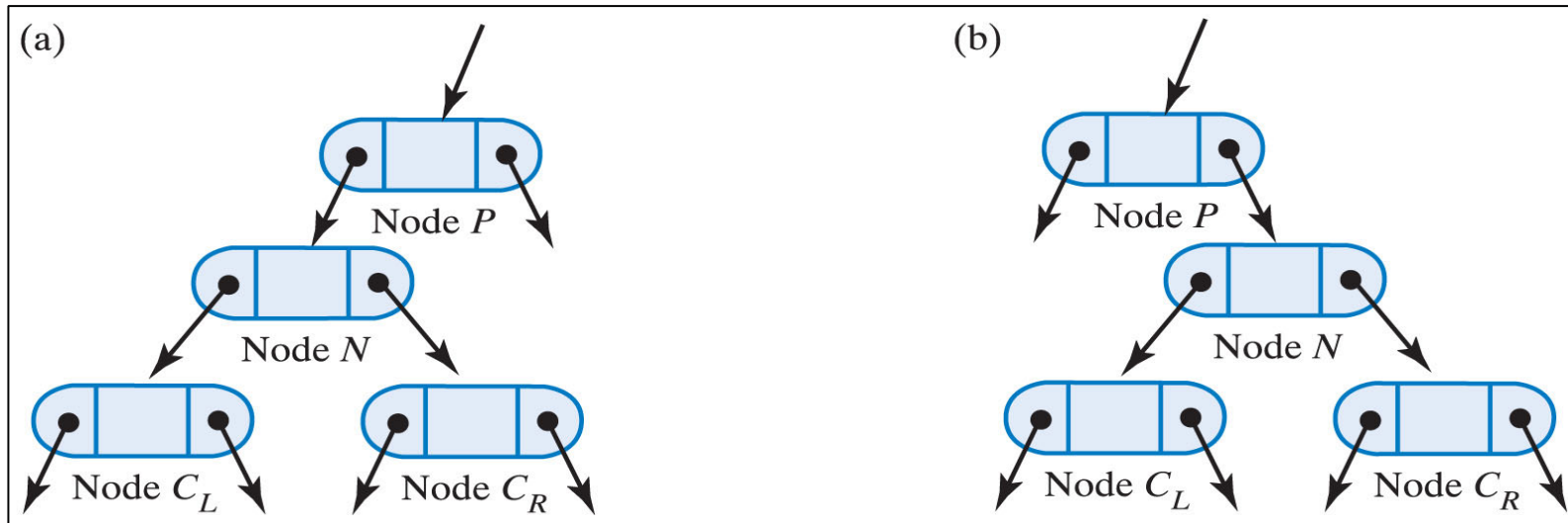
# Remove Entry: One Child



2. Node has one child:  
Delete N; set P's pointer to N's child.



# Remove Entry: Two Children



3. Node has two children:

Won't actually delete Node N,  
but will replace its entry with value of its **inorder predecessor node**.

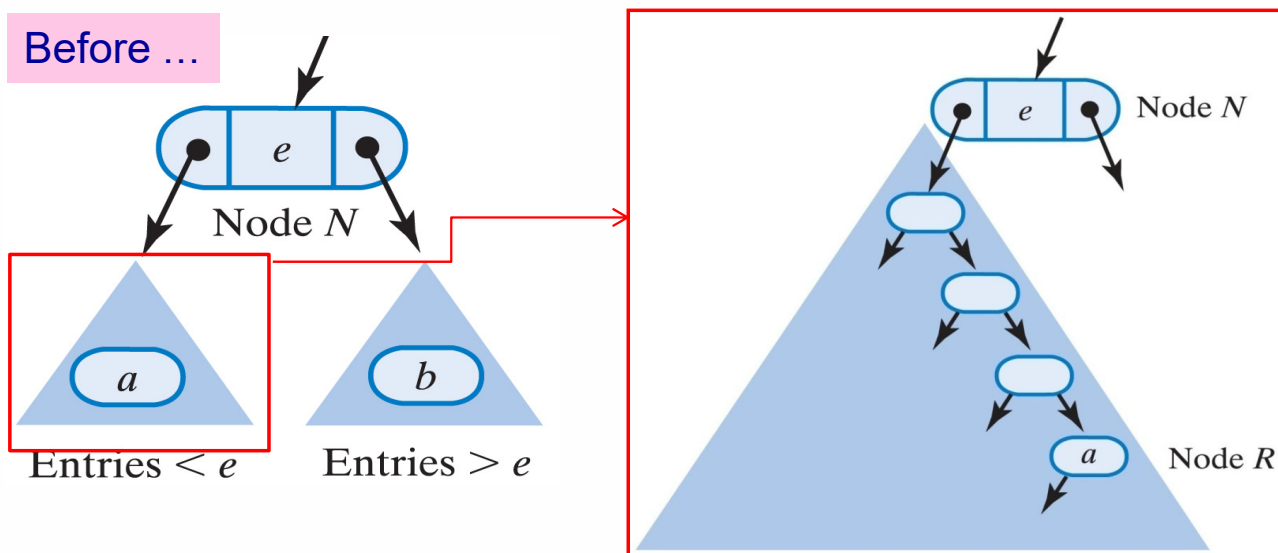
Then delete the inorder predecessor node.

This retains the BST ordering of nodes.





# Remove Entry: Two Children



Node  $N$  has entry  $e$  and two subtrees.  
Inorder sequence:  $a, e, b$   
Node with entry  $a$  is rightmost node ( $R$ ) in left subtree.



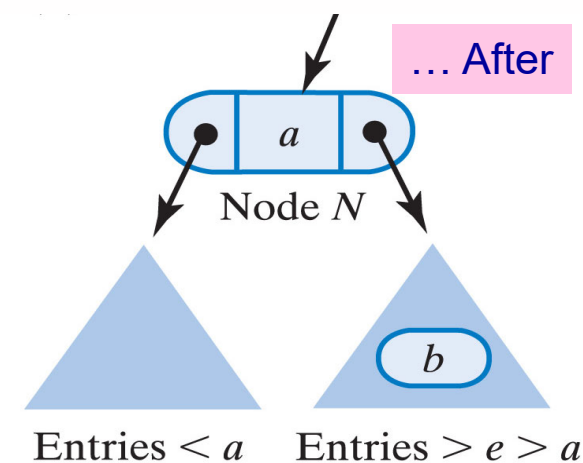
# Remove Entry: Two Children

## Algorithm to Remove Entry With Two Children:

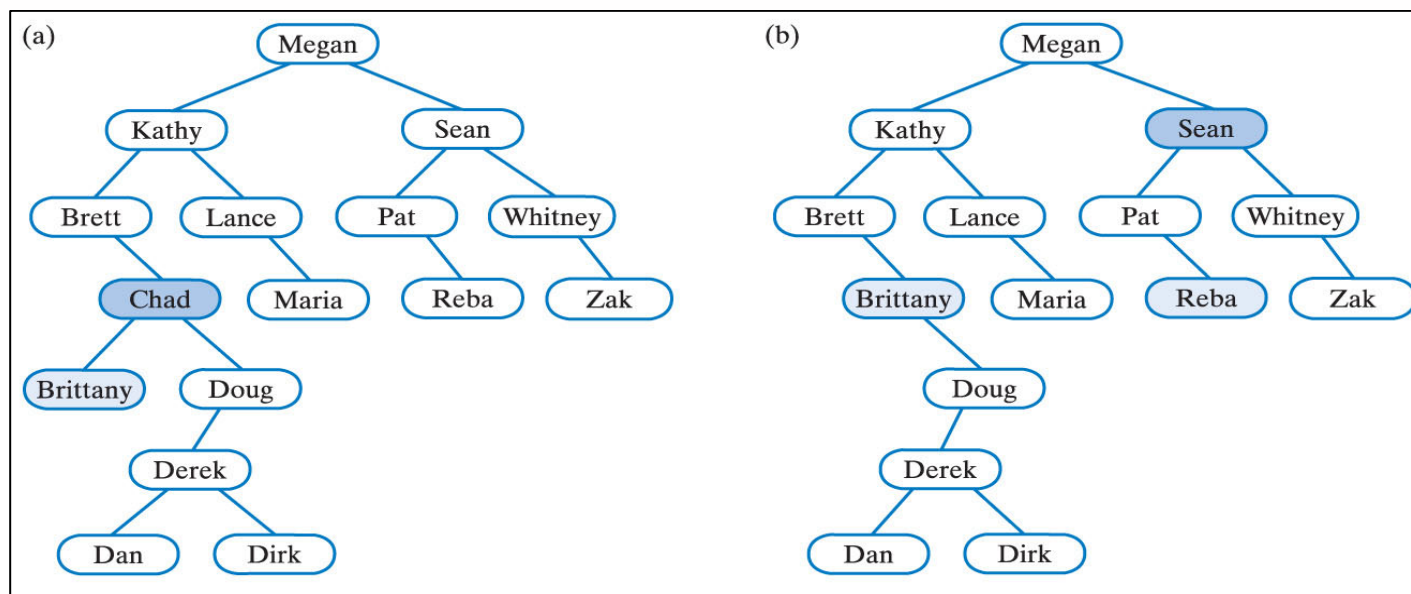
- Initially, Node N to be removed has entry **e**
- Find rightmost node, R, in N's left subtree
- Replace N's entry with entry of R
- Delete Node R

### Note:

Node R may have **no** child or a **left** child;  
Can't have a right child (is rightmost node).



# Remove Entry: Two Children: Examples

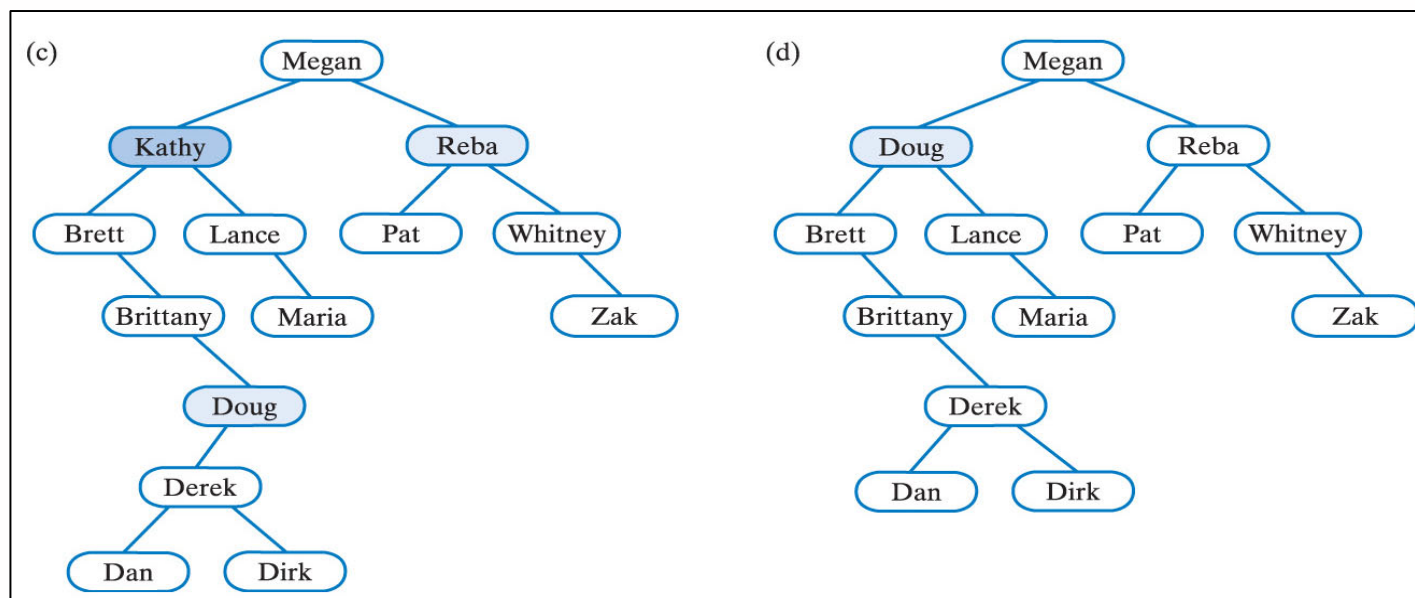


(a) Before removing **Chad**: inorder predecessor is **Brittany**

(b) Before removing **Sean**: inorder predecessor is **Reba**



# Remove Entry: Two Children: Examples



(c) Before removing **Kathy**: inorder predecessor is **Doug**

(d) After removing **Kathy**



# Remove Entry: Root

This is the final case to consider

Root may have no, one or two children

Root has **no** children:

Delete it: tree becomes *empty*

Root has **one** child:

Delete it: child becomes the root

Special case of removing node with one child

Root has **two** children:

Follow the standard procedure for node with two children

Root's entry will be changed, but a different node deleted



# Efficiency of Operations

Operations add, remove, getEntry require a search that begins at the root

Maximum number of comparisons is directly proportional to the height,  $h$  of the tree

These operations are  $O(h)$

Thus, we like to create binary search trees that have the *minimum* height possible

This leads to the idea of [AVL Trees](#) (coming up)



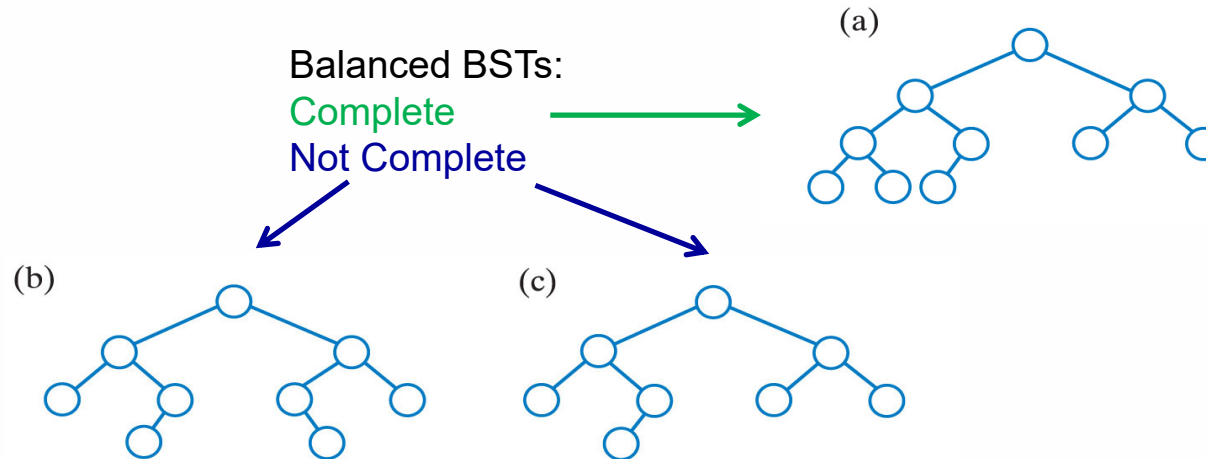
# Importance of Balance

## Balanced Tree:

Subtrees of each node differ in height by **no more than 1**

**Balanced BST** has minimum height for a given number of nodes

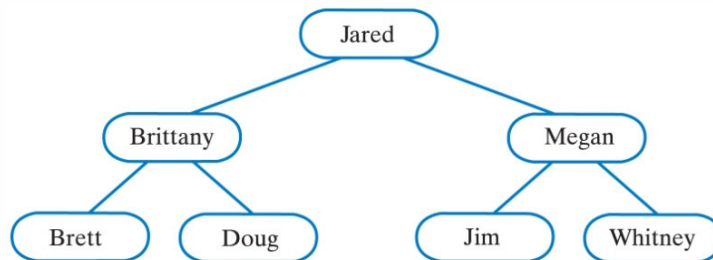
Therefore, operations on it are **as efficient as possible**



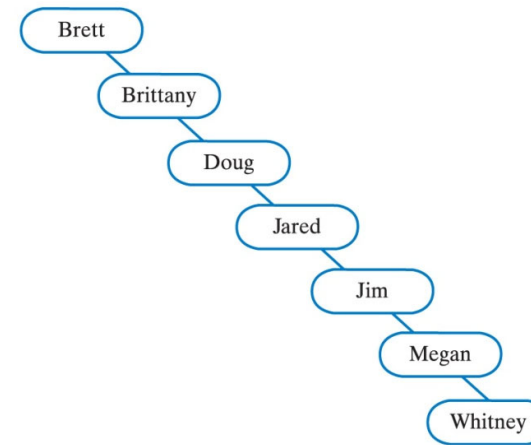
# Balance & Efficiency

What is efficiency of searching Balanced & Unbalanced BSTs of size  $n$  nodes?

Will do in class ...



Balanced



Worst-case Unbalanced





# AVL Trees

An AVL tree is a form of binary search tree

Originally proposed by Adelson-Velskii & Landis

Whenever it becomes unbalanced

Rearranges its nodes to get a balanced binary search tree

Balance of a binary search tree upset when

**Add** a node

**Remove** a node

During these operations, the AVL tree must *rearrange* nodes to maintain balance

These are termed **rotations**

*Note: the unbalanced node is the LOWEST one with a difference of 2 in height between its left & right children*



# AVL Tree: Example

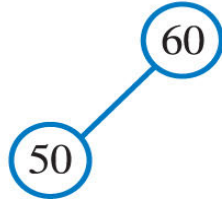
Starting with empty tree, insert:  
60, 50, 20, 80, 90

Insertion of 20 requires a **right rotation** around 60 to re-balance, making its left child into its parent.

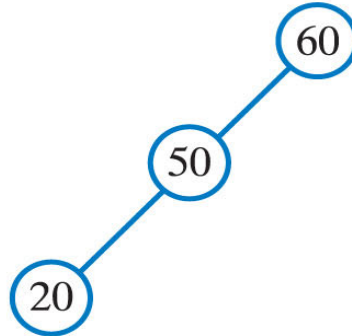
(a)



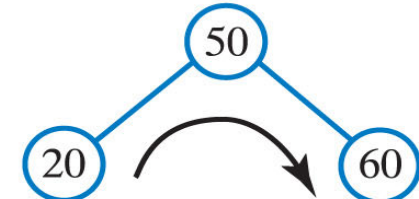
(b)



(c)



(d)

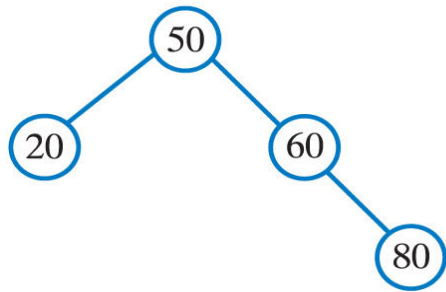


# AVL Tree: Example

Starting with empty tree, insert:  
60, 50, 20, 80, 90

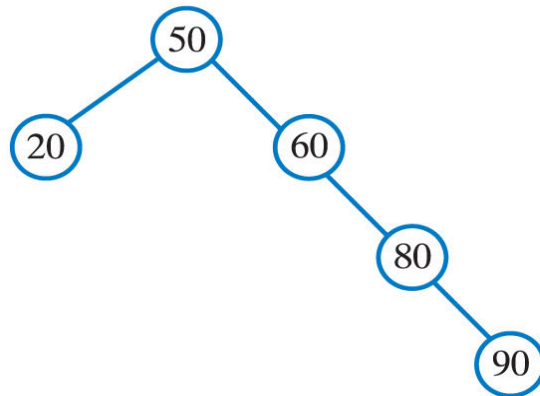
Insertion of 90 requires a **left rotation** around 60 to re-balance, making its right child into its parent.

(a)



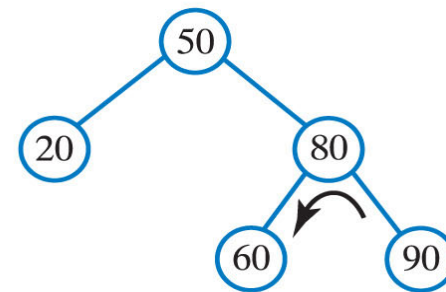
Balanced

(b)



Unbalanced

(c)



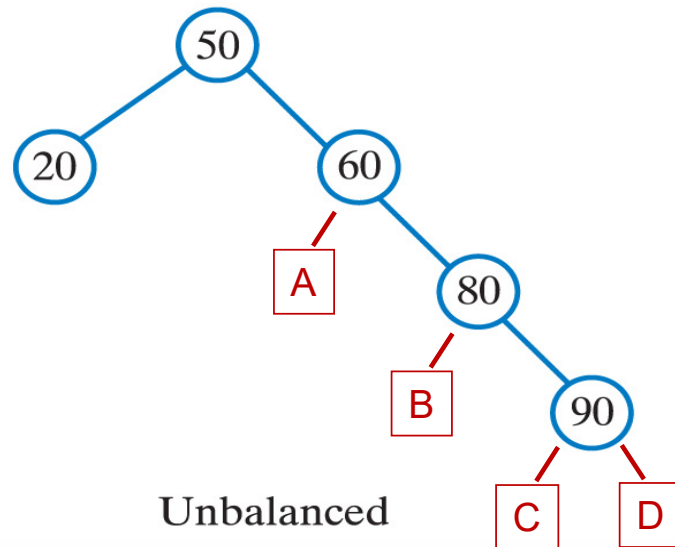
Balanced



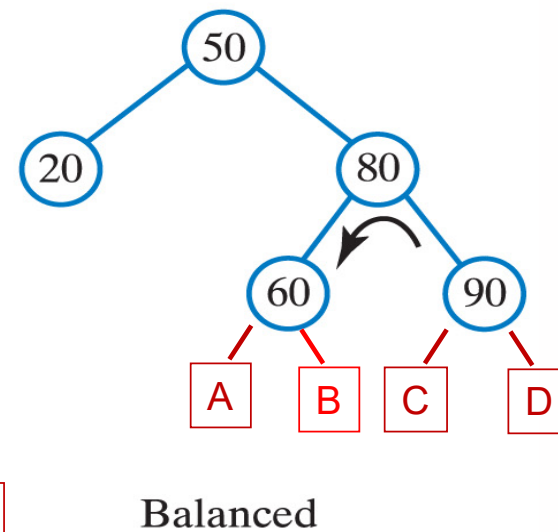
# Rotations: An Important Point

If any of the nodes have subtrees, they get reassigned in such a way as to keep every node with 2 children and to ensure that the in-order traversal is preserved.

(b)



(c)



# Rebalancing an AVL Tree

The general rules ...

An addition [or deletion] can cause a temporary imbalance

Let **N** be the unbalanced node closest to the new leaf

**Right rotation** required if:

Addition occurs in left subtree of N's left child

**Left rotation** required if:

Addition occurs in right subtree of N's right child

**Right-Left rotation** required if:

Addition occurs in left subtree of N's right child

**Left-Right rotation** required if:

Addition occurs in right subtree of N's left child

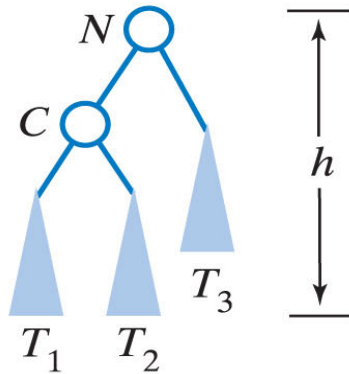
We will look at these cases ...



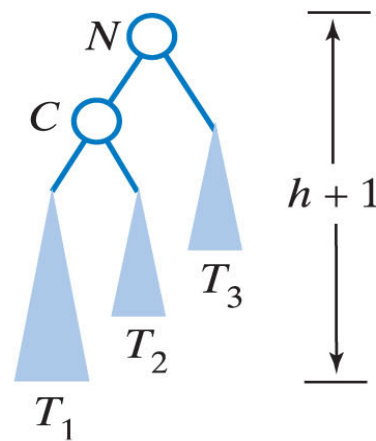
OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# AVL Tree: Single Rotations

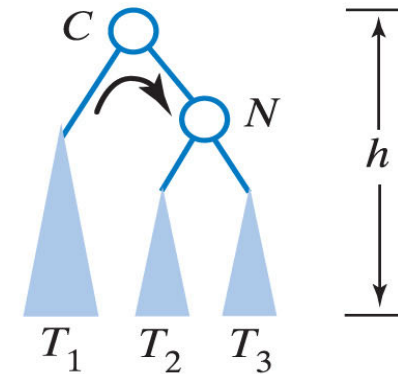
(a) Before addition



(b) After addition



(c) After right rotation

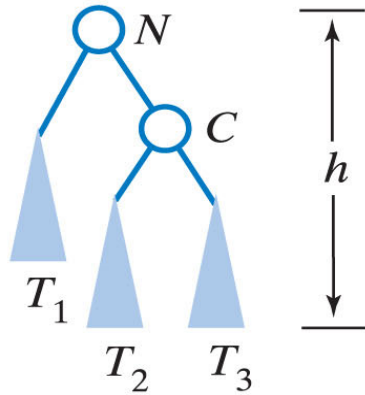


Before and after an addition to a left subtree of left child:  
requires a **right rotation** to maintain its balance.

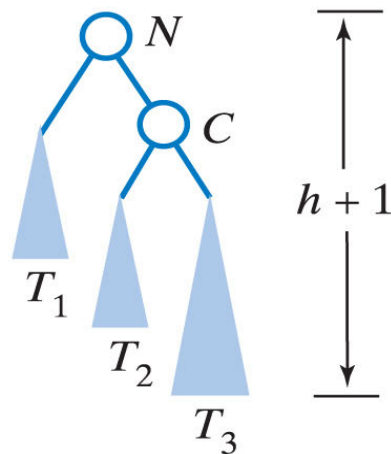


# AVL Tree: Single Rotations

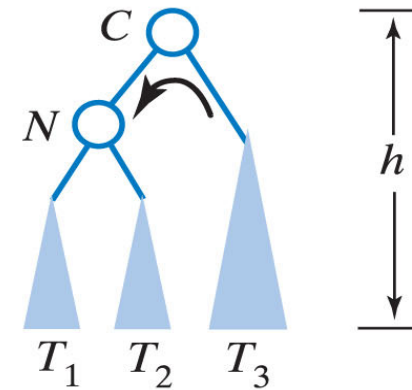
(a) Before addition



(b) After addition



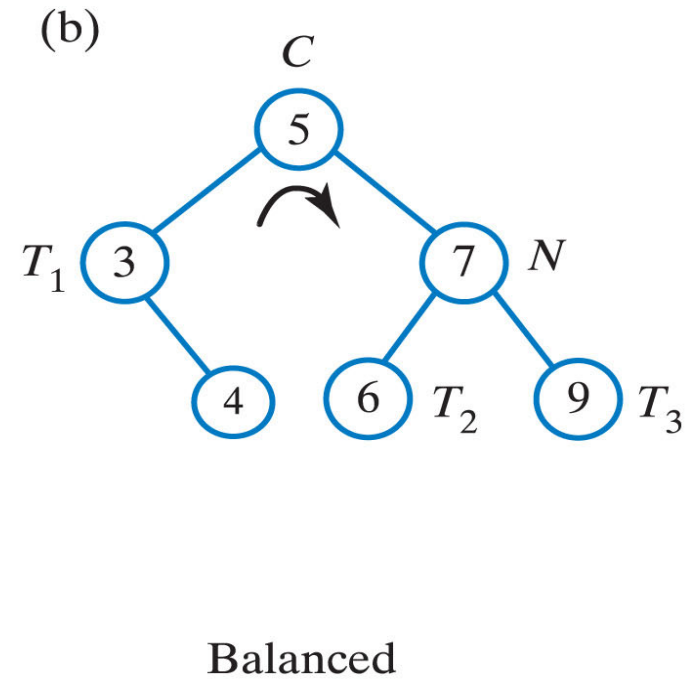
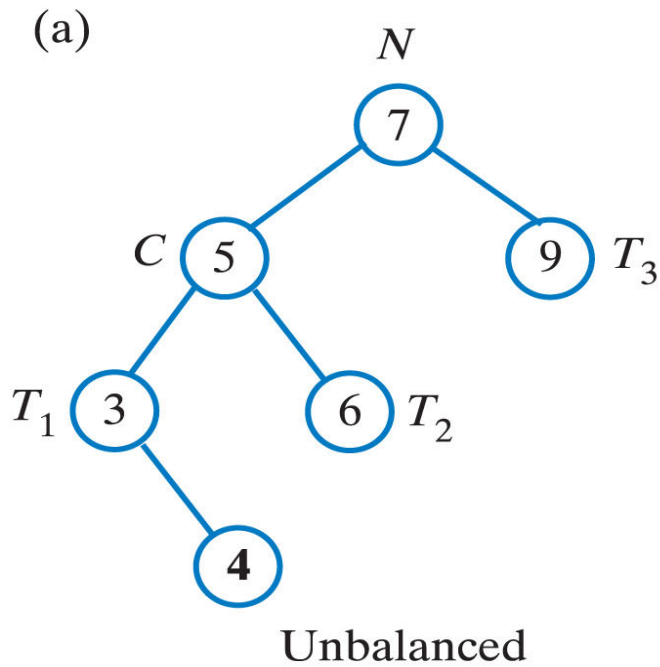
(c) After left rotation



Before and after an addition to right subtree of right child:  
requires a **left rotation** to maintain balance.



# AVL Tree: Single Rotations: Example



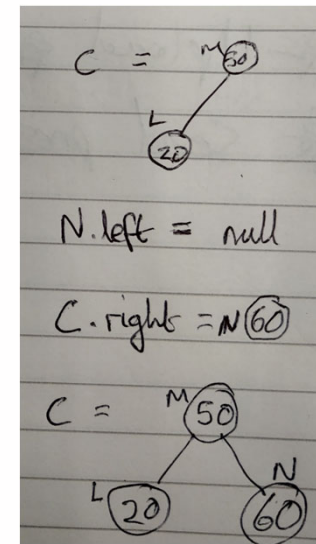
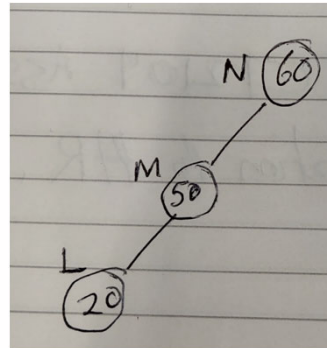
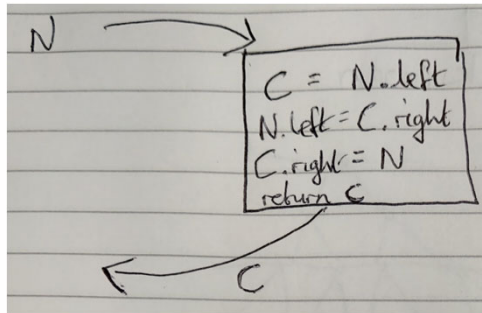
Before and after a **right** rotation restores balance to an AVL tree





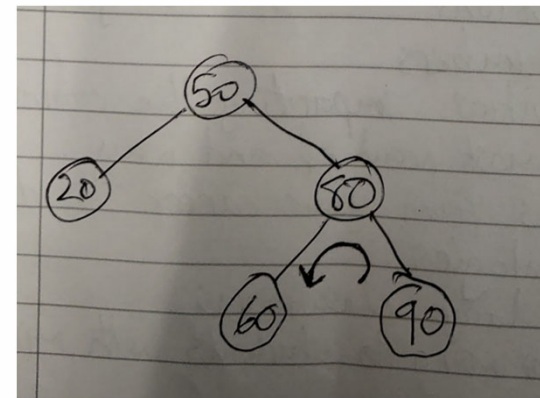
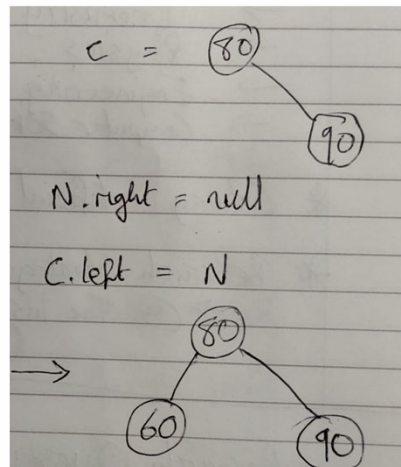
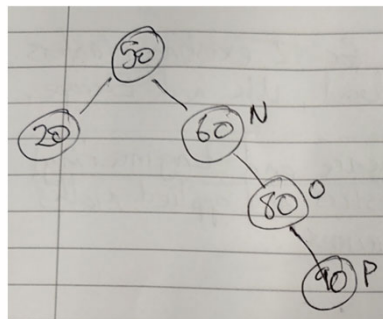
# AVL Tree: Code for Right Rotation

```
BinaryNodeInterface<T> rotateRight(BinaryNodeInterface<T> nodeN)
{
    BinaryNodeInterface<T> nodeC = nodeN.getLeftChild();
    nodeN.setLeftChild(nodeC.getRightChild());
    nodeC.setRightChild(nodeN);
    return nodeC;
}
```



# AVL Tree: Code for Left Rotation

```
BinaryNodeInterface<T> rotateLeft(BinaryNodeInterface<T> nodeN)
{
    BinaryNodeInterface<T> nodeC = nodeN.getRightChild();
    nodeN.setRightChild(nodeC.getLeftChild());
    nodeC.setLeftChild(nodeN);
    return nodeC;
}
```



# Double Rotations

An imbalance at node N of an AVL Tree can be corrected by a double rotation if

The addition occurred in the left subtree of N's right child (**right-left** rotation) or ...

The addition occurred in the right subtree of N's left child (**left-right** rotation)

A double rotation is accomplished by performing two single rotations

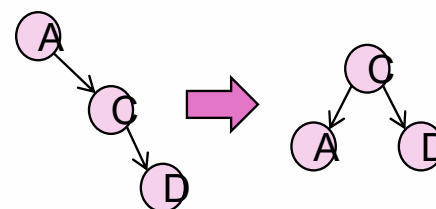
A rotation about N's child [which will change the child]

A rotation about N itself

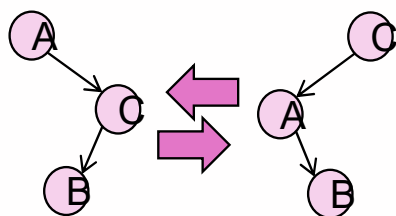


# Double Rotations: Motivation

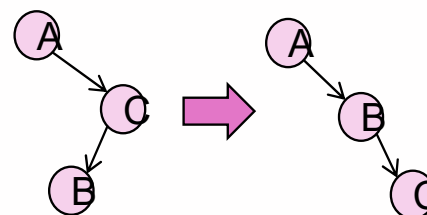
A single rotation can correct imbalances that look like this:



But not this:



The first rotation about the child does this:

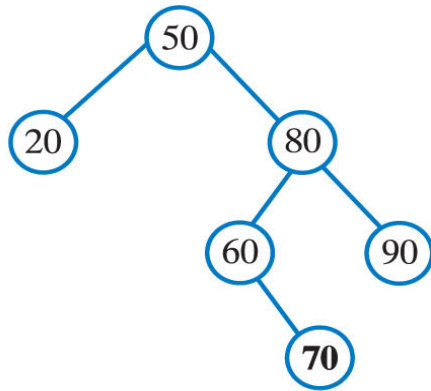


And the second one works as illustrated at the top

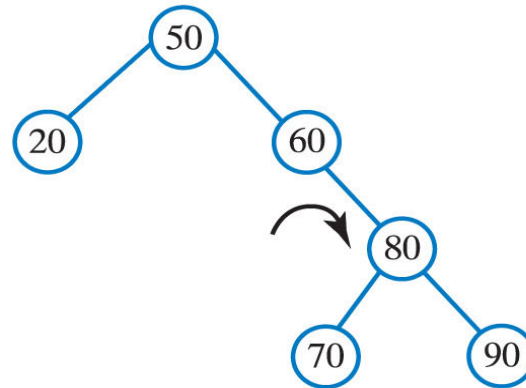


# AVL Tree: Double Rotations

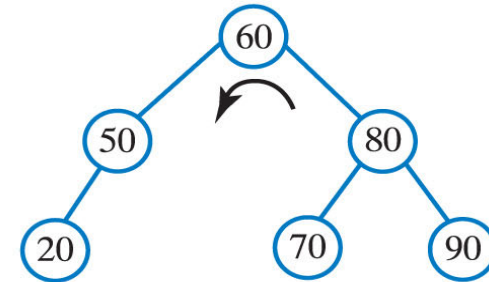
(a) After adding 70



(b) After right rotation



(c) After left rotation



## Example:

Adding 70 to the tree (a) destroys its balance.

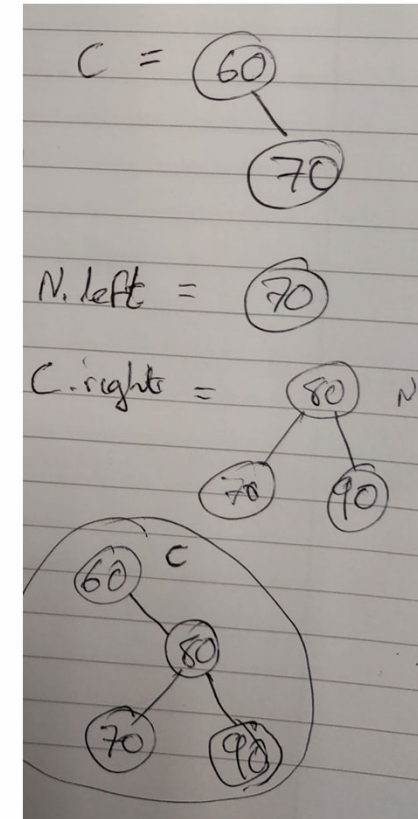
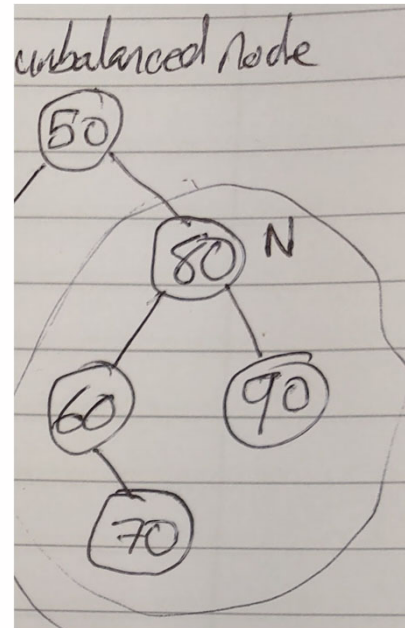
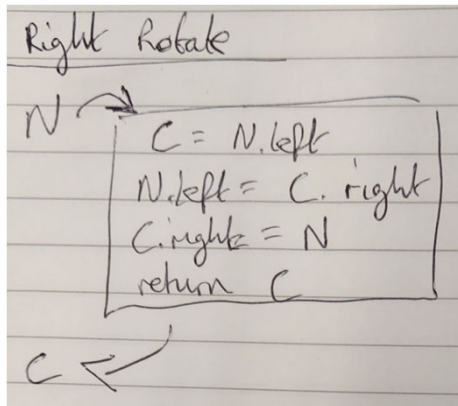
Unbalanced node is 50. To restore the balance, perform:

(b) a right rotation around right child; followed by

(c) a left rotation around 50.



# AVL Tree: Double Rotations



## Example:

Adding 70 to the tree (a) destroys its balance.

Unbalanced node is 50. To restore the balance, perform:

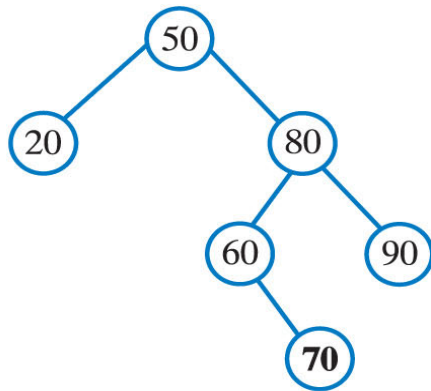
(b) a right rotation around right child; followed by

(c) a left rotation around 50.

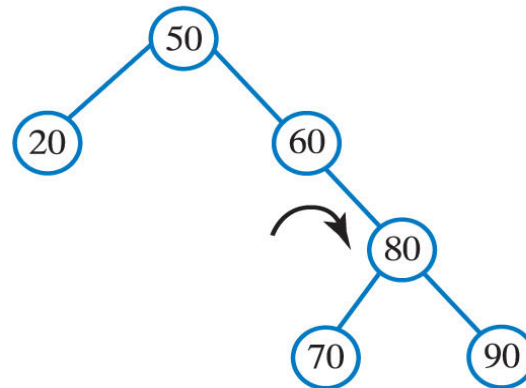


# AVL Tree: Double Rotations

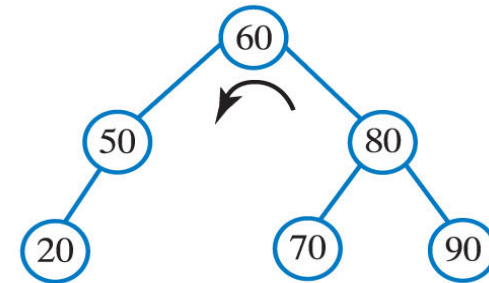
(a) After adding 70



(b) After right rotation



(c) After left rotation



## Example:

Adding 70 to the tree (a) destroys its balance.

Unbalanced node is 50. To restore the balance, perform:

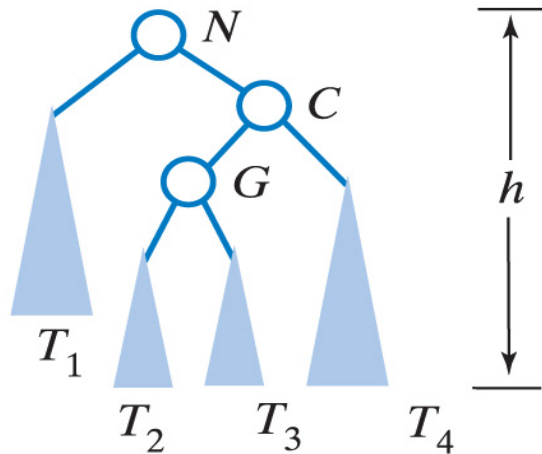
(b) a right rotation around right child; followed by

(c) a left rotation around 50.

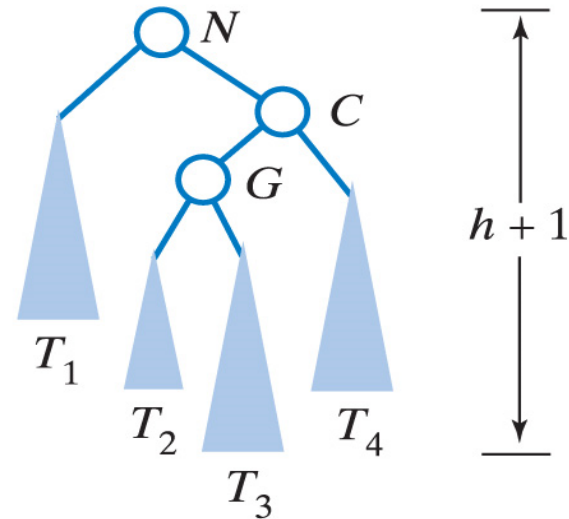


# AVL Tree: Double Rotations

(a) Before addition



(b) After addition



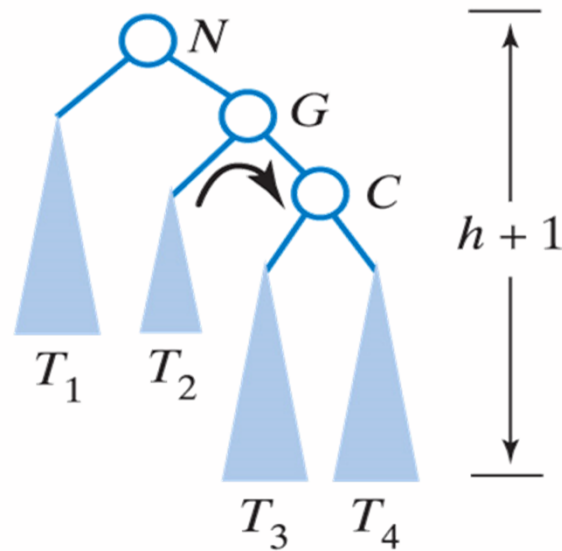
Before and after an addition to an AVL subtree that requires both a right rotation and a left rotation to maintain its balance (see next slide also)



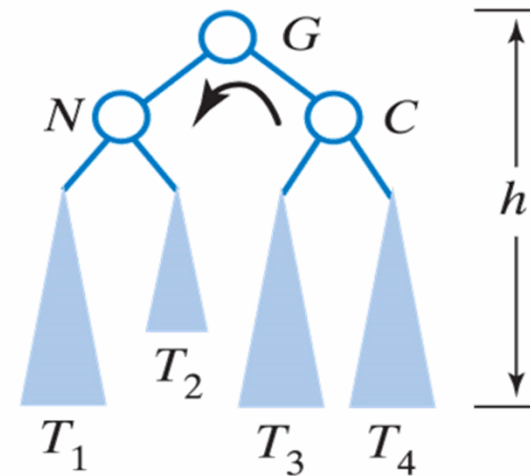


# AVL Tree: Double Rotations

(c) After right rotation



(d) After left rotation



Before and after an addition to an AVL subtree that requires both a right rotation and a left rotation to maintain its balance (from previous slide)



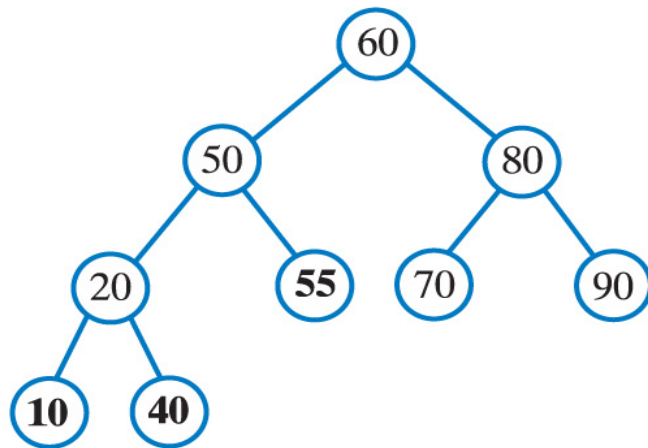
## AVL Tree: Algorithm for Right-Left Rotation

```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node
// nodeN due to an addition
// in the left subtree of nodeNs right child.
nodeC = right child of nodeN
Set nodeNs right child to the node returned
        by rotateRight(nodeC)
return rotateLeft(nodeN)
```

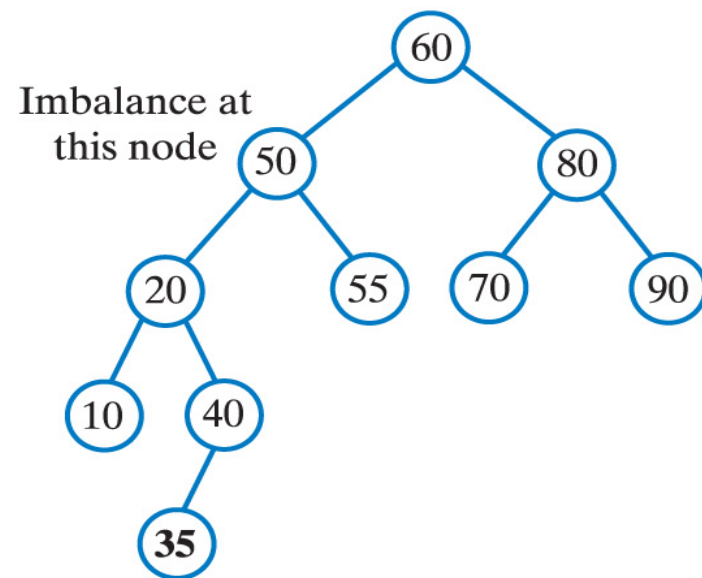


# AVL Tree: Left-Right Rotations Example

(a) After adding 55, 10, and 40



(b) After adding 35



Previous AVL Tree after:

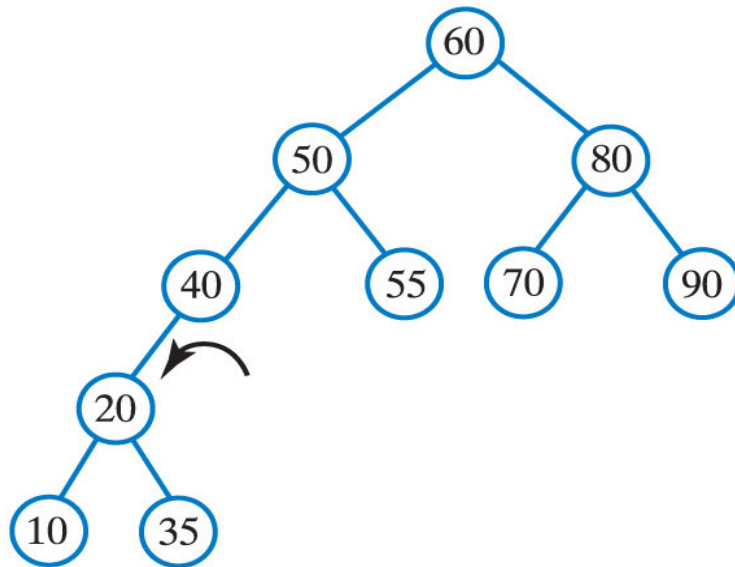
(a) three additions that maintain its balance;

(b) after an addition that destroys the balance (continues ...)

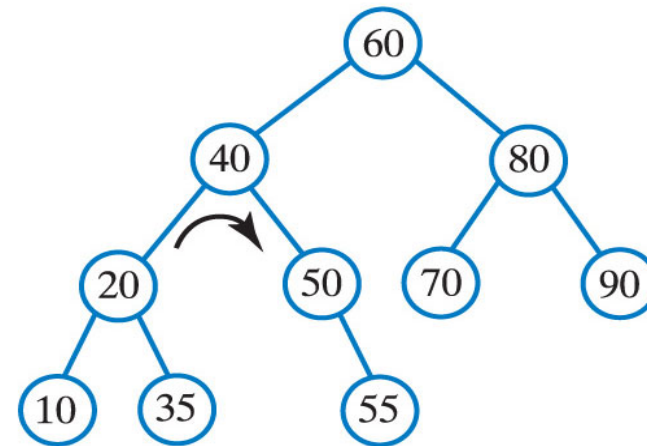


# AVL Tree: Left-Right Rotations Example

(c) After left rotation about 40



(d) After right rotation about 40



(c) after a left rotation around child [20→40]

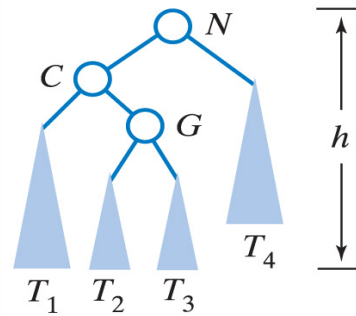
(d) after a right rotation around unbalanced node [50→40]



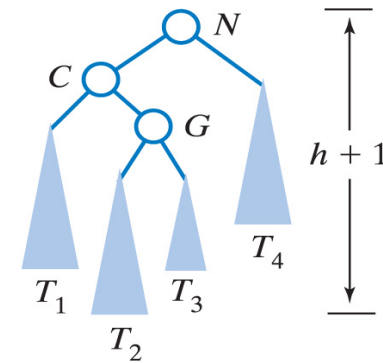
# AVL Tree: Left-Right Rotations

Before and after an addition to an AVL subtree that requires a left and then right rotation to maintain its balance.

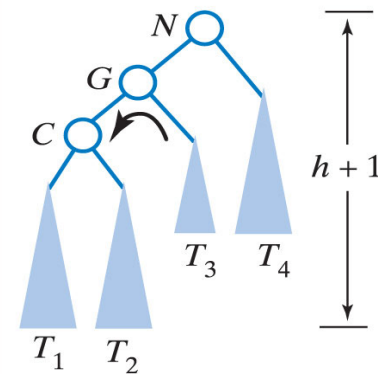
(a) Before addition



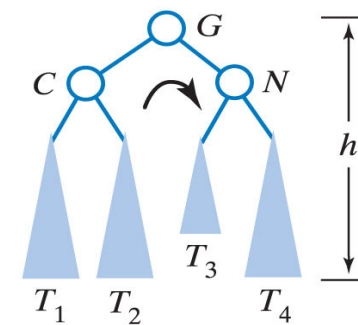
(b) After addition



(c) After left rotation



(d) After right rotation



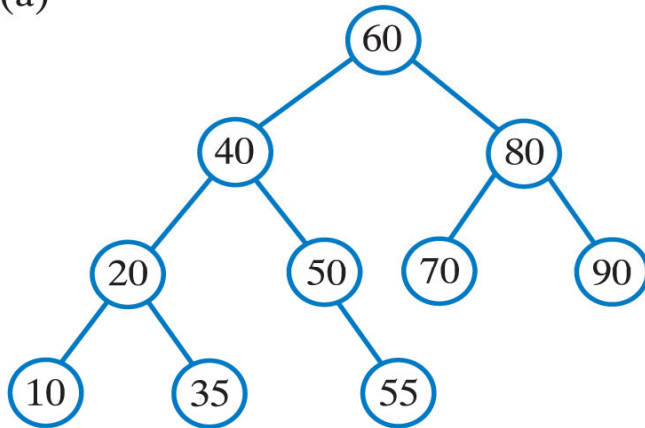
# AVL Tree: Algorithm for Left-Right Rotation

```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given
//node nodeN due to an addition
// in the right subtree of nodeNs left child.
nodeC = left child of nodeN
Set nodeNs left child to the node returned
    by rotateLeft(nodeC)
return rotateRight(nodeN)
```

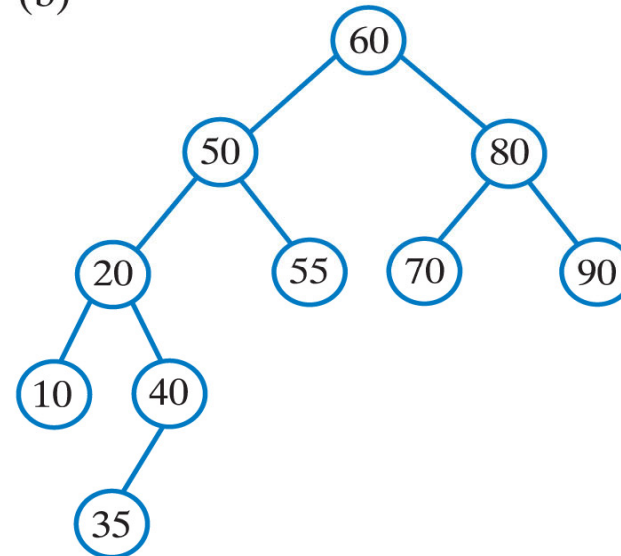


# AVL Tree Compared to BST

(a)



(b)



The result of adding  
60, 50, 20, 80, 90, 70, 55, 10, 40, 35 to initially empty  
(a) AVL Tree; (b) Binary Search Tree.

Both maintain BST structure, but AVL Tree keeps lower depth.



OLLSCOIL NA GAILLIMHIE  
UNIVERSITY OF GALWAY

School of Computer Science

# Other Tree Structures

## k-node:

A node that has  $k$  children and can store  $k-1$  data items:

E.g. **2-node**: 2 children, 1 data item (standard BT node)

E.g. **3-node**: 3 children, 2 data items

## 2-3 Tree:

Balanced search tree with 2-nodes and 3-nodes

Specific rules for splitting 3-nodes when extra data added

## 2-4 Tree:

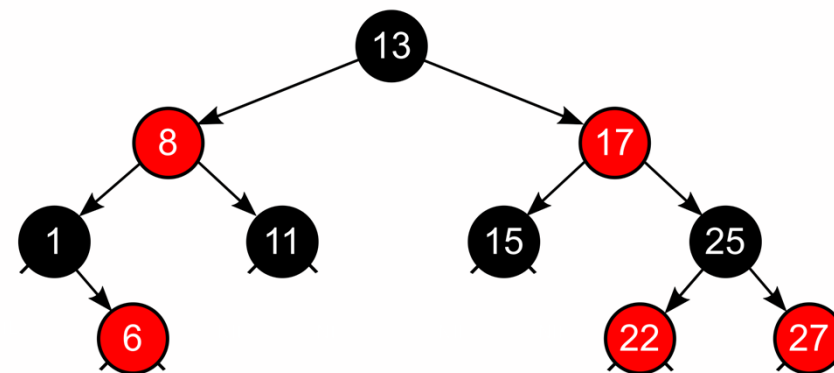
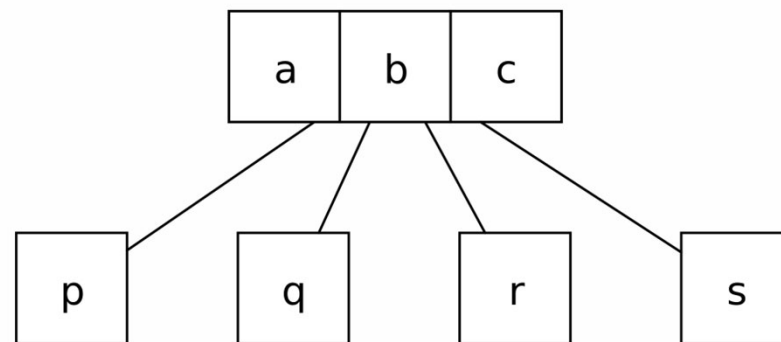
Balanced search tree with 2-nodes, 3-nodes, 4-nodes

Easier to maintain balance than AVL Tree

## Red-Black Tree

A binary tree that is logically equivalent to 2-4 Tree

Ease of maintaining balance but efficiency of binary nodes





# What You Achieved in This Topic

- Explain the structure and use of Binary Search Trees, including algorithms to process them
- Analyse complexity of Search Tree algorithms
- Discuss the distinctions between balanced and unbalanced Search Trees, and how balance is achieved
- Describe and use AVL Trees, including algorithms to operate on them
- List and discuss other forms of Search Tree
- Understand how to implement these data structures in Java and demonstrate how to use them





OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# CT2109

## OOP: Data Structures and Algorithms



**Dr. Frank Glavin**  
Room 404, CS Building  
[Frank.Glavin@UniversityofGalway.ie](mailto:Frank.Glavin@UniversityofGalway.ie)  
School of Computer Science

University  
ofGalway.ie

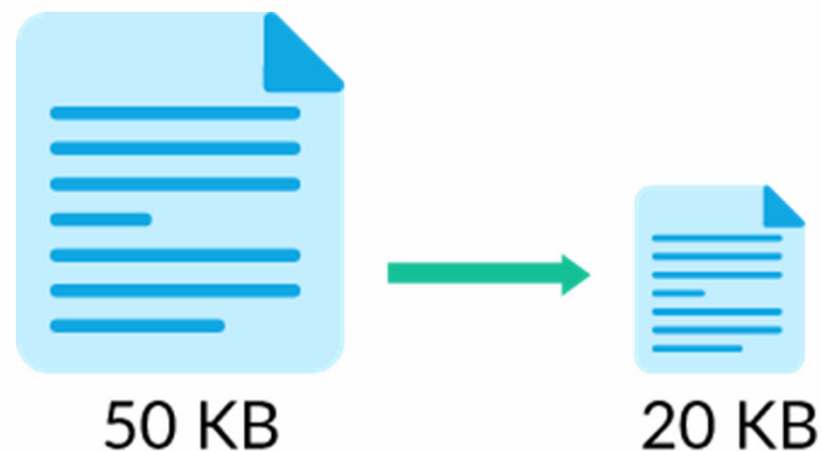
# What You'll Achieve in This Topic

- Explain the concepts of lossy and lossless compression
- Describe an algorithm for run-length encoding
- Describe the Huffman encoding algorithm, including the algorithms and data structures used in it
- Demonstrate the application of run-length encoding and Huffman encoding to compressing data such as text



# Data Compression & Terminology

- Important part of data storage & transmission
- Bitstream:
  - Data is stored/transmitted in binary form
  - Stream of bits may be a file or a message
- Lossy compression:
  - Data size is reduced, but some information is lost
  - *Is this ever reasonable?*
  - *Example?*
- Lossless compression:
  - No data is lost
  - Compression is **reversible** to recover original bitstream
  - *Example?*



# Data Compression

Simple example:

"aaaaabbbbbfff" is a string

"6a6b3f" is a simple compressed representation:

Notes:

This is a simple form of *run-length encoding*

Introduces new symbols to describe original sequence

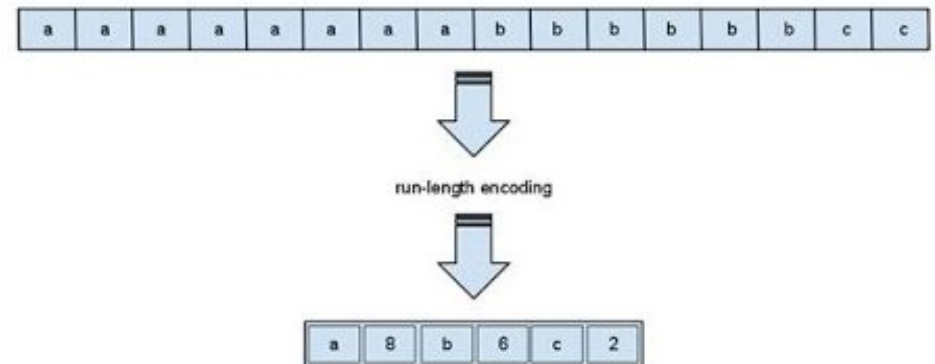
Original sequence had 15 chars, new one has 6 chars

Can be reversed to recover original string

What about these sequences?

"abbaafbafbbaafb"

"aabbfaabbfaabbf"



# Can Every Bitstream Be Compressed?

Assuming **lossless** compression,  
is it always possible to make a bitstream smaller?

What do you think?



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Can Every Bitstream Be Compressed?

Assuming lossless compression (i.e. reversible),  
is it always possible to make a bitstream smaller?

No!

Proof by *contradiction*:

Assume such an algorithm exists

After applying algorithm, reapply it to resulting stream

Continue until length is 1: impossible to reverse

Proof by *counting*:

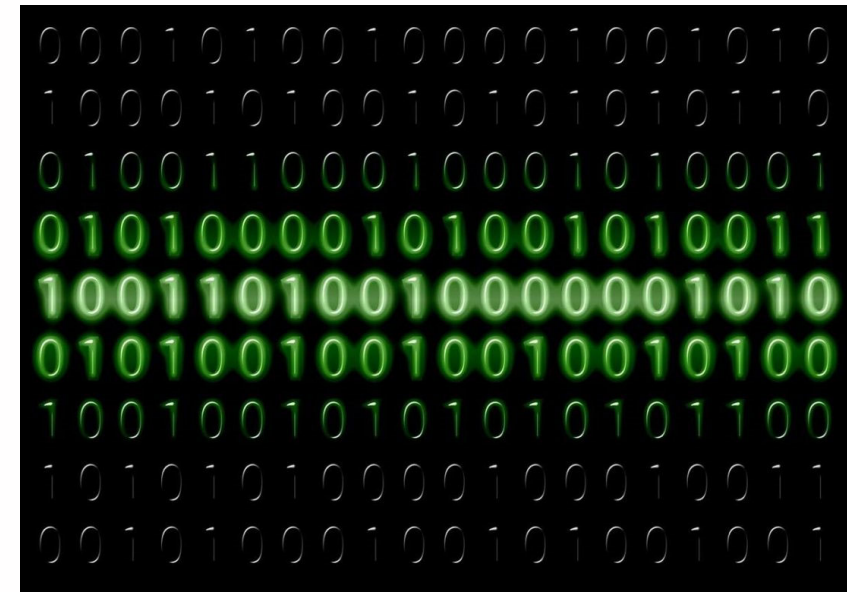
Assume bitstream of length  $N$

There are  $2^N$  different such bitstreams

There are only  $2^N - 1$  bitstreams of length shorter than  $N$

=> will be at least one *collision*

=> Cannot be a reversible mapping between these



OLLSCOIL NA GAILLIMH  
UNIVERSITY OF GALWAY

# Can Every Bitstream Be Compressed?

To be compressible, bitstream must have structure/ regularities that can be *summarized*

Amenable to compression:

## Natural text data

Contains frequent words that may appear often

Some letters appear with high frequency, but all letters have same length encoding

XML is routinely compressed

## Binary image data

Blocks of single colours:

long runs of 1, 0





# Can Every Bitstream Be Compressed?

Not amenable to compression

Random data

Data that is already compressed

Try this at home (optional!)

Write program to output a random binary stream, GZIP it

Does the file size reduce much or at all?

What about if you output a random **text** stream?

Compressibility relates to **entropy**

Amount of disorder in a data stream

High entropy: low compressibility

Note on GZIP

Combines **LZ77** (a dictionary encoder) & **Huffman** coding



OLLSCOIL NA GAILLIMHE  
UNIVERSITY OF GALWAY

# Run-Length Encoding (1)

A simple compression method

Example bitstream:

0000000000001111111100000000011111111111

12 **0**s + 8 **1**s + 9 **0**s + 11 **1**s

Method

Encode this as the numbers of alternating **0**s and **1**s

Always begin with the number of **0**s (which might be *none*)

Assume we use 4 bits to encode each number

Result:

1100|1000|1001|1011

12=1100, 8=1000, 9=1001, 11=1011

Compression Ratio =  $16/40 = 40\%$



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Run-Length Encoding (2)

## Algorithm issues

How many bits should be used to store the counts?

What do we do if a run is too long?

What about runs that are shorter than the corresponding encoding?

## Standard choices

Use 8 bits (runs are between 0 and 255 in length)

If a run is longer than 255, insert a run of length 0

(300 **1**s is encoded as 255 **1**s + 0 **0**s + 45 **1**s

Encode short runs, even if this lengthens the output



# Run-Length Encoding (3)

Popular for bitmaps

If a bitmap's resolution is doubled:

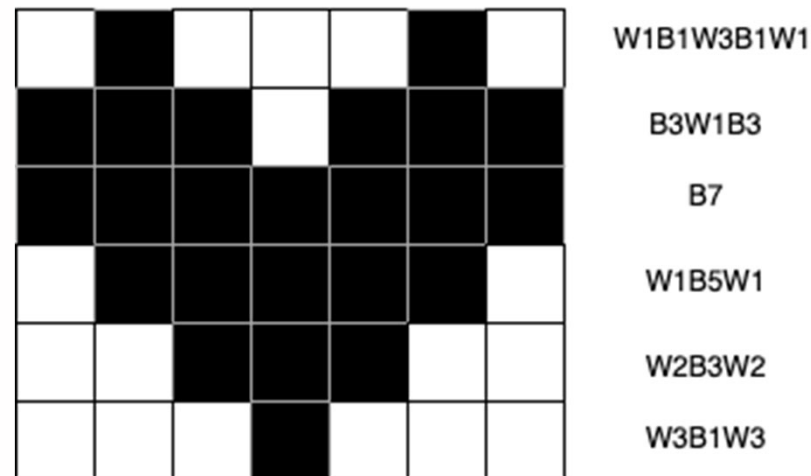
bitstream increases x4

RLE compressed version typically increases x2

Works in a single pass:

No need to look ahead when compressing or decompressing

Example of Image compression using RLE



# Run-Length Encoding: Limitations

If bitstream has large numbers of short runs

RLE encoded version can be **longer** than the original!

Worst case: 10101010

*How would this be encoded?*

Natural text contains few repeated letters

Traditional binary representation is 7-bit or 8-bit ASCII

This tends to contain short runs also:

8-bit run encoding is definitely too long;

4-bit run encoding does not do much for it either



# ASCII Binary Representation (Excerpt)

48	0	00110000	67	C	01000011	86	V	01010110	105	i	01101001
49	1	00110001	68	D	01000100	87	W	01010111	106	j	01101010
50	2	00110010	69	E	01000101	88	X	01011000	107	k	01101011
51	3	00110011	70	F	01000110	89	Y	01011001	108	l	01101100
52	4	00110100	71	G	01000111	90	Z	01011010	109	m	01101101
53	5	00110101	72	H	01001000	91	[	01011011	110	n	01101110
54	6	00110110	73	I	01001001	92	\	01011100	111	o	01101111
55	7	00110111	74	J	01001010	93	]	01011101	112	p	01110000
56	8	00111000	75	K	01001011	94	^	01011110	113	q	01110001
57	9	00111001	76	L	01001100	95	_	01011111	114	r	01110010
58	:	00111010	77	M	01001101	96	`	01100000	115	s	01110011
59	;	00111011	78	N	01001110	97	a	01100001	116	t	01110100
60	<	00111100	79	O	01001111	98	b	01100010	117	u	01110101
61	=	00111101	80	P	01010000	99	c	01100011	118	v	01110110
62	>	00111110	81	Q	01010001	100	d	01100100	119	w	01110111
63	?	00111111	82	R	01010010	101	e	01100101	120	x	01111000
64	@	01000000	83	S	01010011	102	f	01100110	121	y	01111001
65	A	01000001	84	T	01010100	103	g	01100111	122	z	01111010
66	B	01000010	85	U	01010101	104	h	01101000	123	{	01111011



# 8-Bit ASCII Encoding

Consider the text “Mississippi”

8-bit ASCII:

```
01001101011010010111001101110011011010010111
00110111001101101001011100000111000001101001
```

Every character encoded with 8 bits

88 bits

RLE will not compress it well

How can we do better?

Do we need all 8 bits?

Yes, in general case if we use all ASCII characters

Do all characters need a full 8 bits to encode them?

$2^8-1$  possibilities if we used all seqs of length 1-7

M	01001101
i	01101001
s	01110011
s	01110011
i	01101001
s	01110011
s	01110011
i	01101001
p	01110000
p	01110000
i	01101001



# Huffman Encoding (1)

Main idea: variable-length codes

Use short-length codes for more frequently occurring characters

These should be **prefix-free**

If we used a simple encoding such as A=0, B=1, C=10, ..., would not know if 10 encoded "BA" or "C"

We want codes to be uniquely decodable without needing any delimiters or prefixes  
(Fixed-length codes like 8-bit ASCII are also prefix-free)





# Huffman Encoding (2)

To come up with a prefix-free variable encoding,  
we construct a Huffman Tree:

This is a binary tree that will give us the final encoding

See next slides

Using the tree, read off variable-length codes for each character

Encode this message using the tree

Note: we need to store the encoding as well as the message being encoded

Reduces compression ratio a lot for short messages

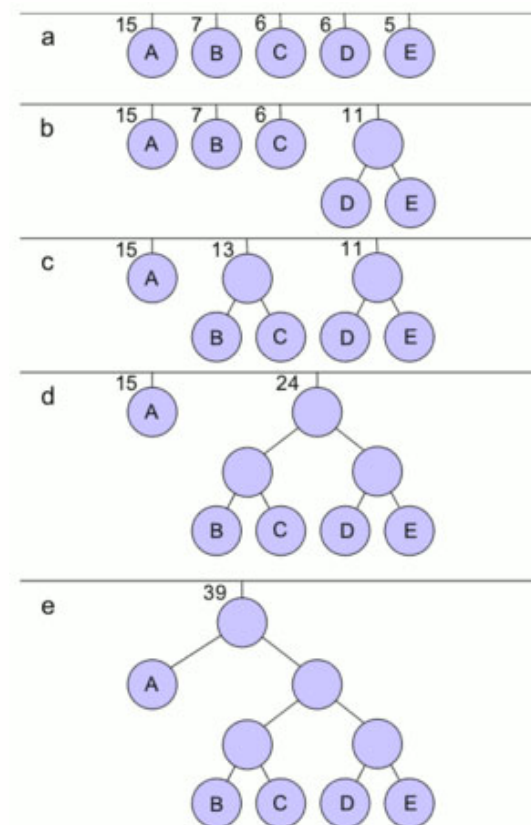
Can use an agreed encoding, e.g. for English text

With an agreed encoding, computation time of constructing the tree is also avoided



# Huffman Encoding – Example (1)

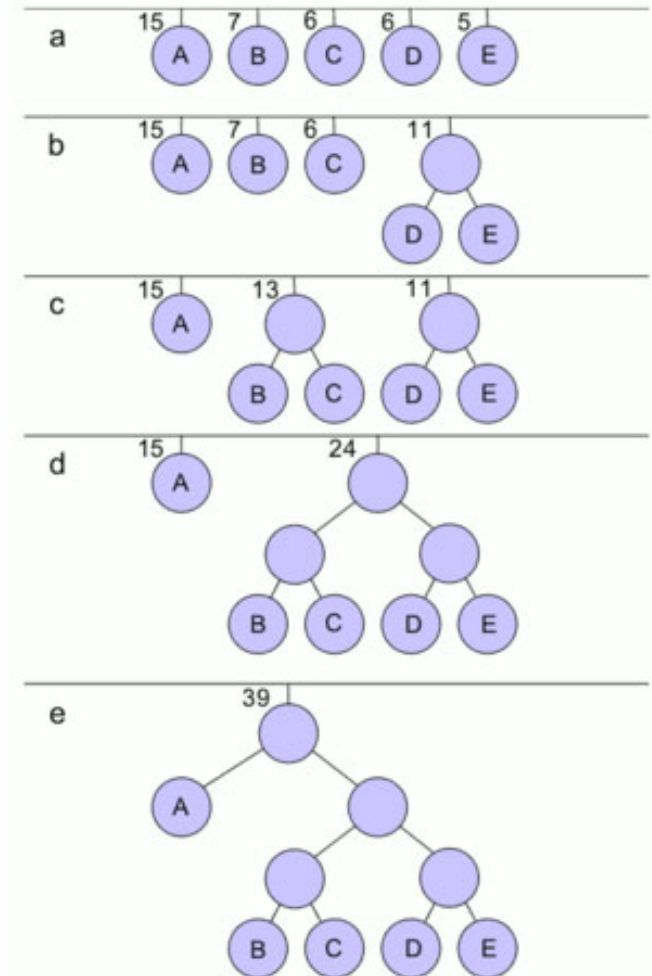
1. Create a set of trees, each consisting of one leaf; each leaf represents a symbol
2. Remove the two trees with the lowest probability, merge them into a single tree, sum up their probabilities and return the new tree into the pool
3. Repeat step 2 until a single tree is left over
4. Generate code words as seen before



# Huffman Encoding – Example (2)

In the example on the right the following code is generated:

Symbol	Code
A	0
B	100
C	101
D	110
E	111



# Algorithm to Construct Huffman Tree Using a Priority Queue

1. Count frequencies of all letters
2. Put them as nodes in a Priority Queue, with *lowest* count having *highest* priority  
A queue where items are inserted according to priority, not at the end; dequeued as normal from front  
In case of a tie, put more recently enqueued item after older items
3. While there are at least 2 nodes on the queue:  
Dequeue the 2 nodes at the front  
Make a new node with them as its 2 children  
Value of new node = sum of children's counts  
Enqueue this new node



# Let's Do It ...

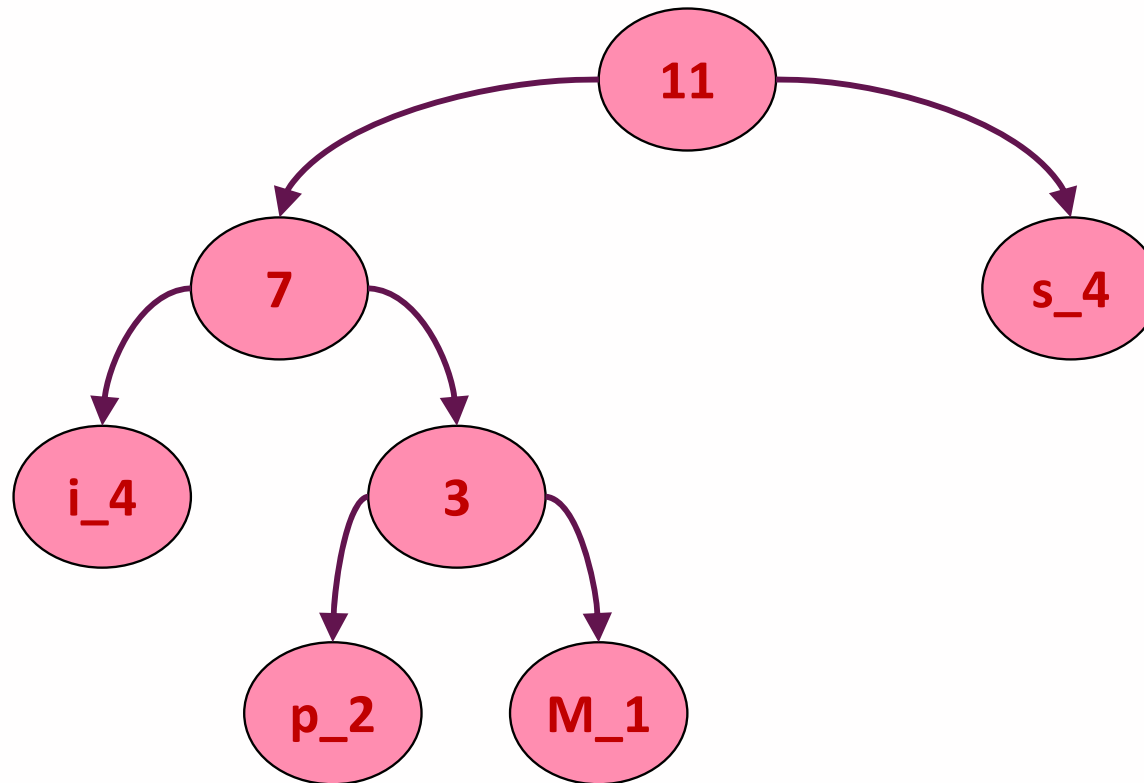
M:1 p:2 i:4 s:4

Each node, apart from the root, represents a bit in the Huffman code:  
each left child is a 0 and each right child is a 1



OLLSCOIL NA GAILLIMH  
UNIVERSITY OF GALWAY

# One Possible Result



# Resultant Encoding

With this, "Mississippi"  
encodes to:

100110011001110110111

Just 21 bits

Excluding the code,

Compression Ratio = 23.9%

Space Savings =  $1 - CR = 76.1\%$

Can you decode this unambiguously?

For decoding, can use the code  
directly, or use the tree:

Start at root

0:left, 1:right, until a leaf is reached

s	0
i	11
p	101
M	100

M	100
i	11
s	0
s	0
i	11
s	0
s	0
i	11
p	101
p	101
i	11



OLLSCOIL NA GAILLIMHÉ  
UNIVERSITY OF GALWAY

# Aside: Relationship to Machine Learning

## Compression relates to Machine Learning:

Goal is to find descriptions (hypotheses) that partially/fully describe a larger set of data

Usually the hypotheses are expressed differently than in zip algorithms

E.g. rules; equations; graphs

Almost always **lossy**: focus on main features of data, not every possible detail

Find hypotheses to fit data through **heuristic search**

Standard ZIP algorithms have been used as similarity metrics for large documents





# What You Achieved in This Topic

- Explain the concepts of lossy and lossless compression
- Describe an algorithm for run-length encoding
- Describe the Huffman encoding algorithm, including the algorithms and data structures used in it
- Demonstrate the application of run-length encoding and Huffman encoding to compressing data such as text

