CSC108: Introduction to Computer Programming

Lecture 11

Wael Aboulsaadat

Acknowledgment: these slides are based on material by: Velian Pandeliev, Diane Horton, Michael Samozi, Jennifer Campbell, and Paul Gries from CS UoT

Recursion

Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.

- Have you had a teacher tell you that you can't use a word in its own definition? This is a *circular* definition.

- In mathematics, recursion is frequently used. The most common example is the factorial:

  For example, $5! = 5(4)(3)(2)(1)$,

  or

  $5! = 5(4!)$

Recursive Definitions

- In other words, $n! = n(n-1)!$

- Or

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

- This definition says that 0! is 1, while the factorial of any other number is that number times the factorial of one less than that number.

Recursive Definitions

- Our definition is recursive, but definitely <u>not</u> circular. Consider 4!
  - 4! = 4(4-1)! = 4(3!)
  - What is 3!? We apply the definition again
    4! = 4(3!) = 4[3(3-1)!] = 4(3)(2!)
  - And so on…
    4! = 4(3!) = 4(3)(2!)
    = 4(3)(2)(1!)
    = 4(3)(2)(1)(0!)
    = 4(3)(2)(1)(1)
    = 24

Recursive Definitions

- Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1. This is called a *base case* for the recursion.

- When the base case is encountered, we get a closed expression that can be directly computed.

Recursive Definitions

- All good recursive definitions have these two key characteristics:
  - There are one or more base cases for which no recursion is applied.
  - One or more recursive case which eventually end up at one of the base cases.

- The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.

Recursive Definitions

- We've seen previously that factorial can be calculated using a loop accumulator.

- If factorial is written as a separate function:

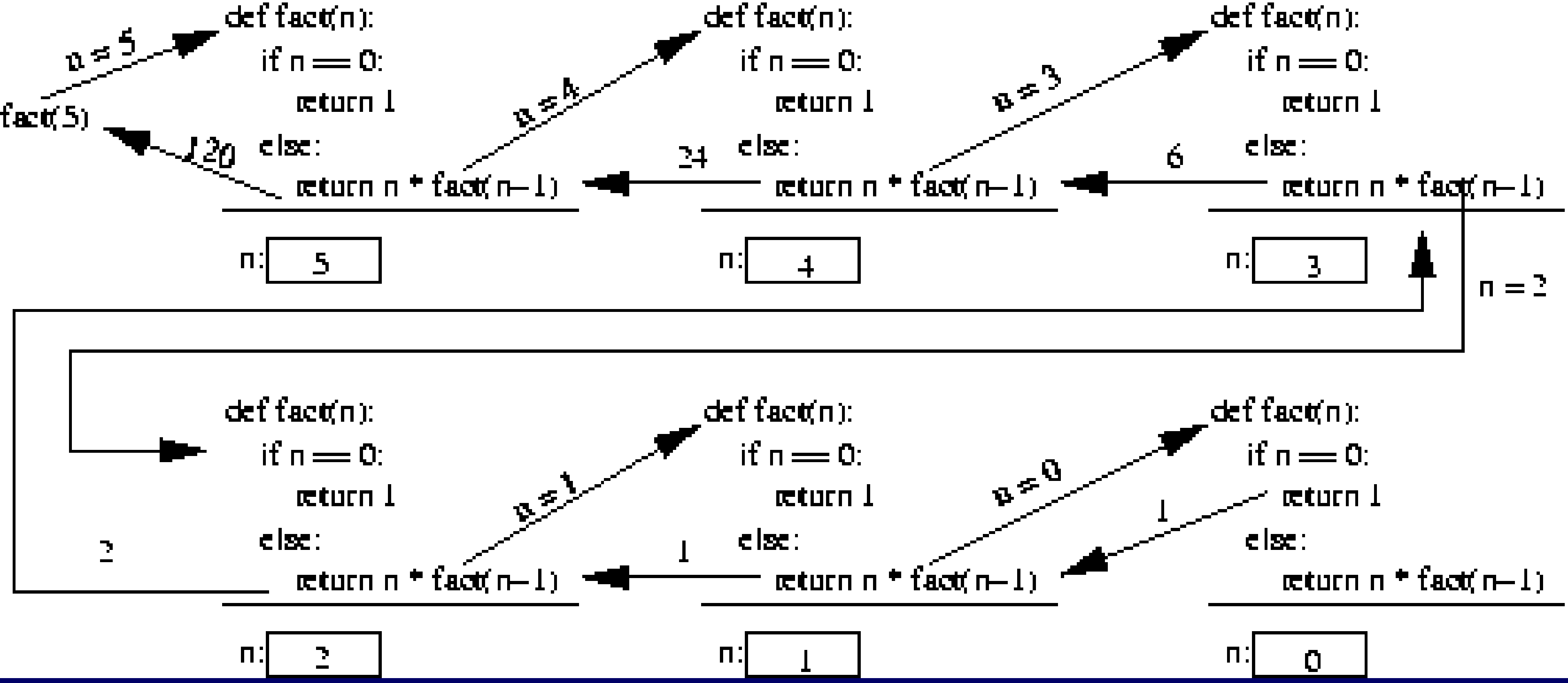
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Recursive Definitions

- We've written a function that calls *itself*, a *recursive function*.

- The function first checks to see if we're at the base case (`n==0`). If so, return 1. Otherwise, return the result of multiplying *n* by the factorial of *n-1*, `fact(n-1)`.

Recursive Definitions



Functions
(revisited)

First-Class Objects

- The term '**first-class object' refers to an object** which has the following important properties:

  - can be stored in variables and data structures

  - can be passed to and returned by a function

- In all programming languages, primitive data types (ints, floats, strings, etc.) are first-class since they conform to the above rules.

- In many of those, functions are not first-class. However, in Python functions are in fact first-class.

- What does this entail?

Representing Functions

- We've alluded to the fact that Python stores function names the same way it stores variable names.
- In a namespace, all Python really has is names that are
- connected to memory addresses (also called pointers).
- For immutable data, those addresses point to primitive data objects like 4. For mutable data, they point to more complex objects, which may have pointers of their own
- For functions, addresses point to the memory space where the low-level commands of the function are stored.

Functions as Variables

- Moreover, it turns out that functions are in fact objects, meaning that they can be manipulated as such.

- For instance we can assign function names to other variables:

```
def even(i):
    return i % 2 == 0

print even(3)
not_odd = even
print not_odd(3)
```

Functions as Parameters

- We can pass function names to other functions as parameters:

```
def add(i,j):
        return i + j
def multiply(i,j):
        return i * j
add(3,5)                # returns 8
multiply(3,5)           # returns 15
def do(fxn, a, b):
        return fxn(a,b)

do(add,3,5)             # calls add, returns 8
```

Functions as First-Class Objects

- This is a varsity-level feature of Python that few programming languages share.
- Using functions as first-class objects (passing them around, renaming them, etc.) affords us some great flexibility when writing code.
- For instance, let's write a function that times the execution of a single-argument function and returns the time it took in seconds.

A Function Timer

```
import time
def runtime(f, arg):
'''f is a 1-argument function. arg is a suitable argument for f. Return the amount of time it takes to run f on arg.'''
    before = time.clock()
    f(arg)
    after = time.clock()
    return (after - before)
```

Using the Timer

- For example, if we wanted to time this function to find which is faster of bubble or selection sort

- We could use:

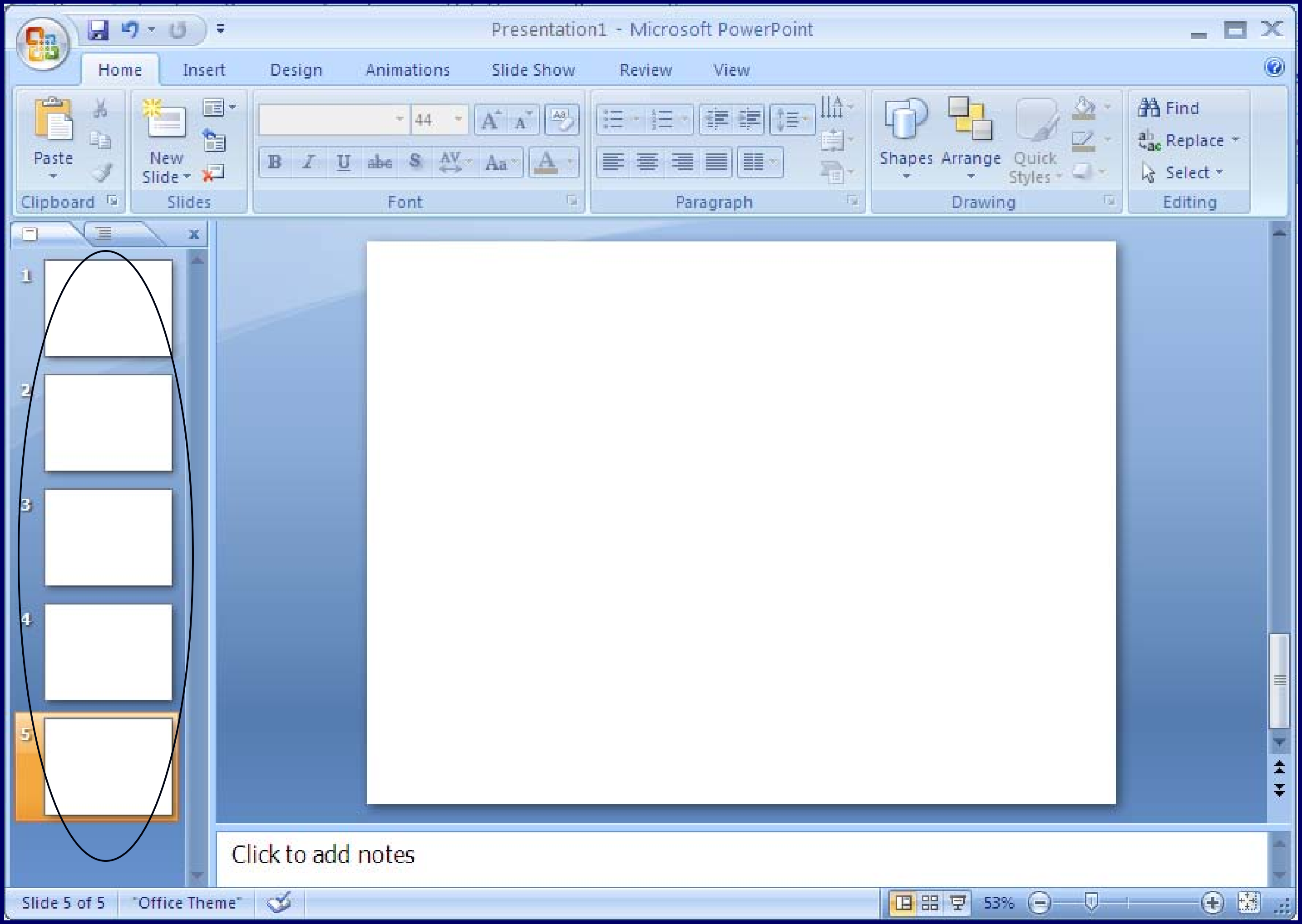
```
print "%1.20f" % runtime(bubble_sort,[3,6,5,4,1])
print "%1.20f" % runtime(selection_sort,[3,6,5,4,1])
```

Data Structures (revisited)

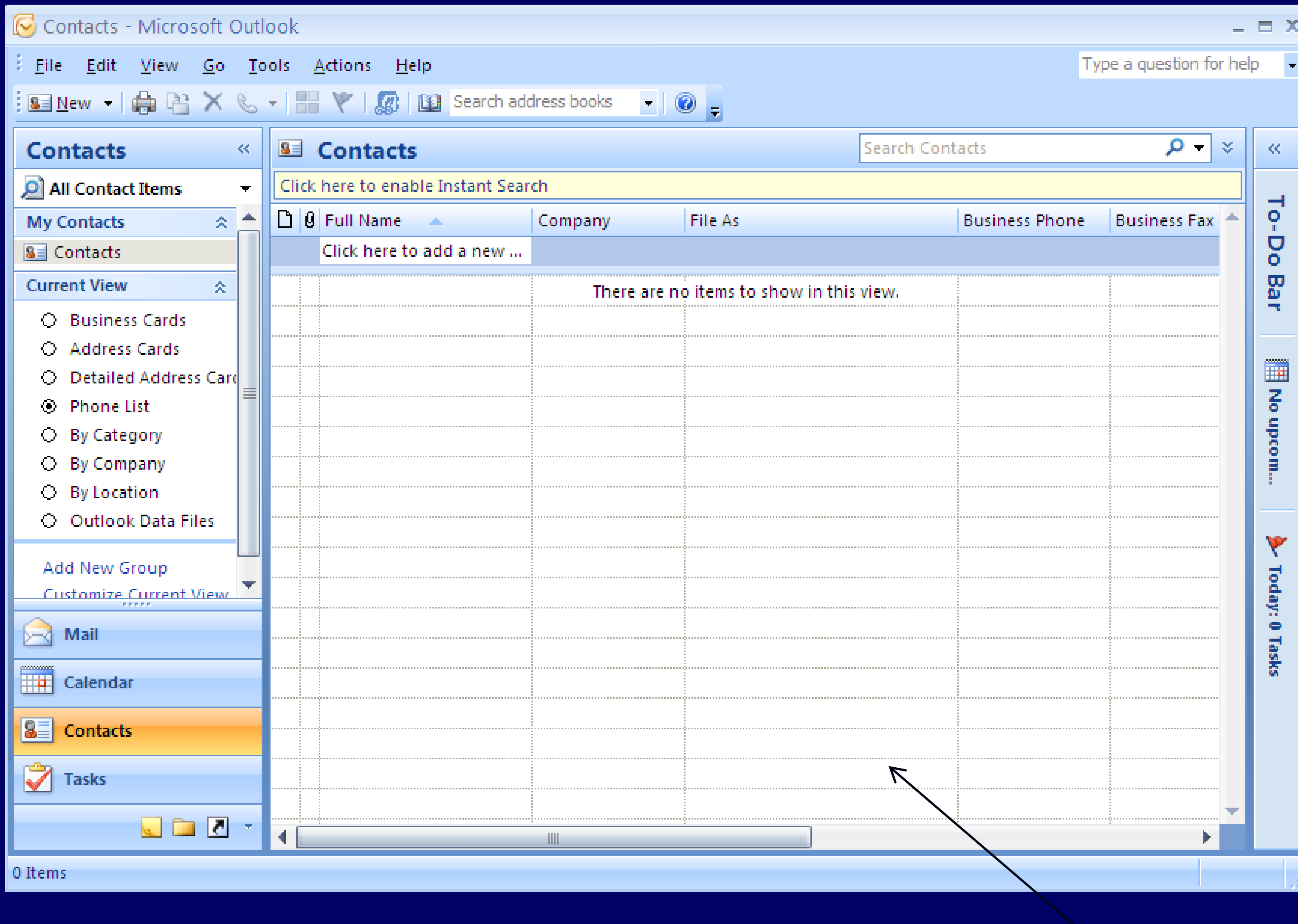
Data structures

- We have learnt the following data structures:
  - List
  - Tuple
  - Dictionary
- The Type of the data structure you use is dependent on the functionality required.

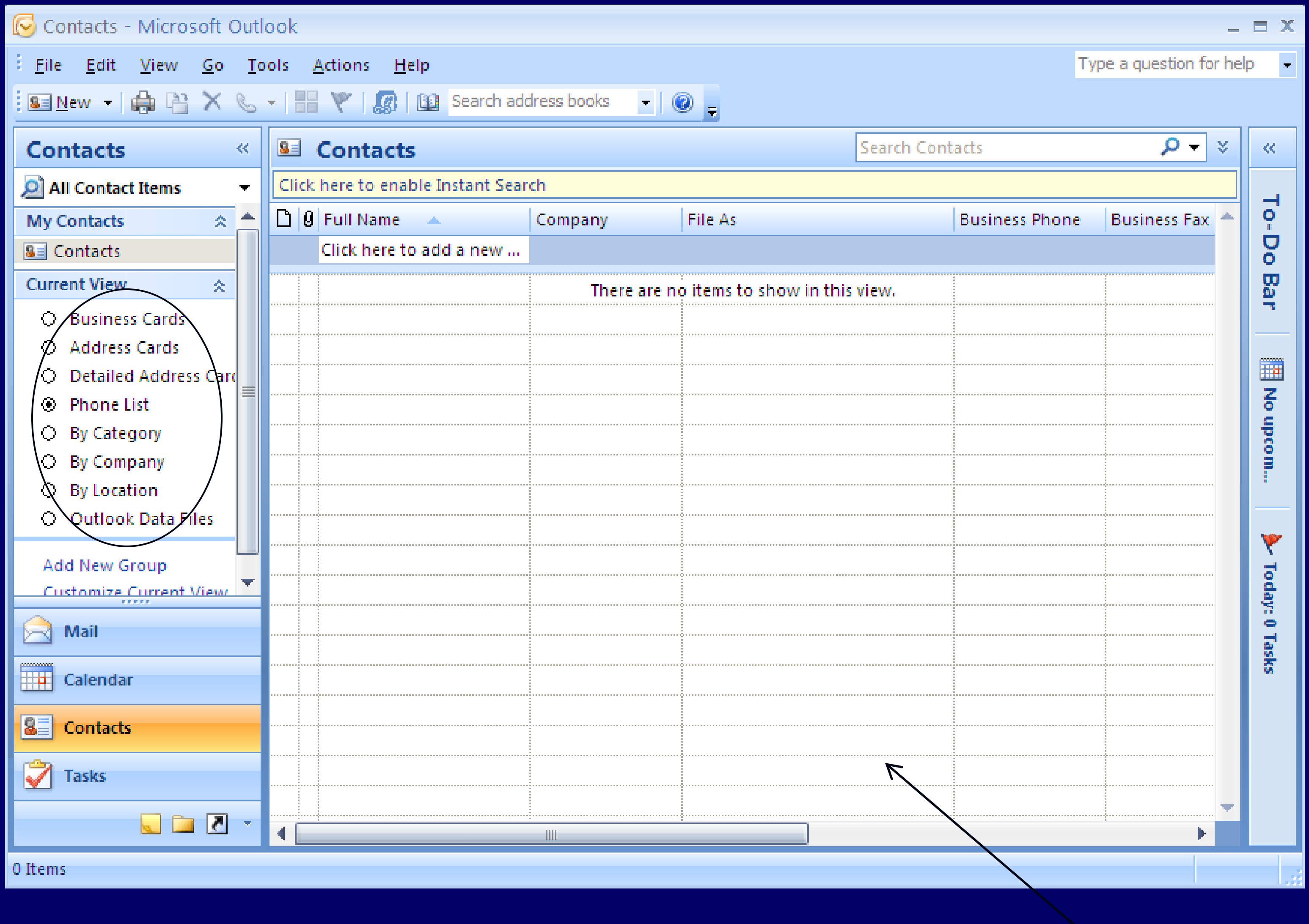
Which data structure should be used here?



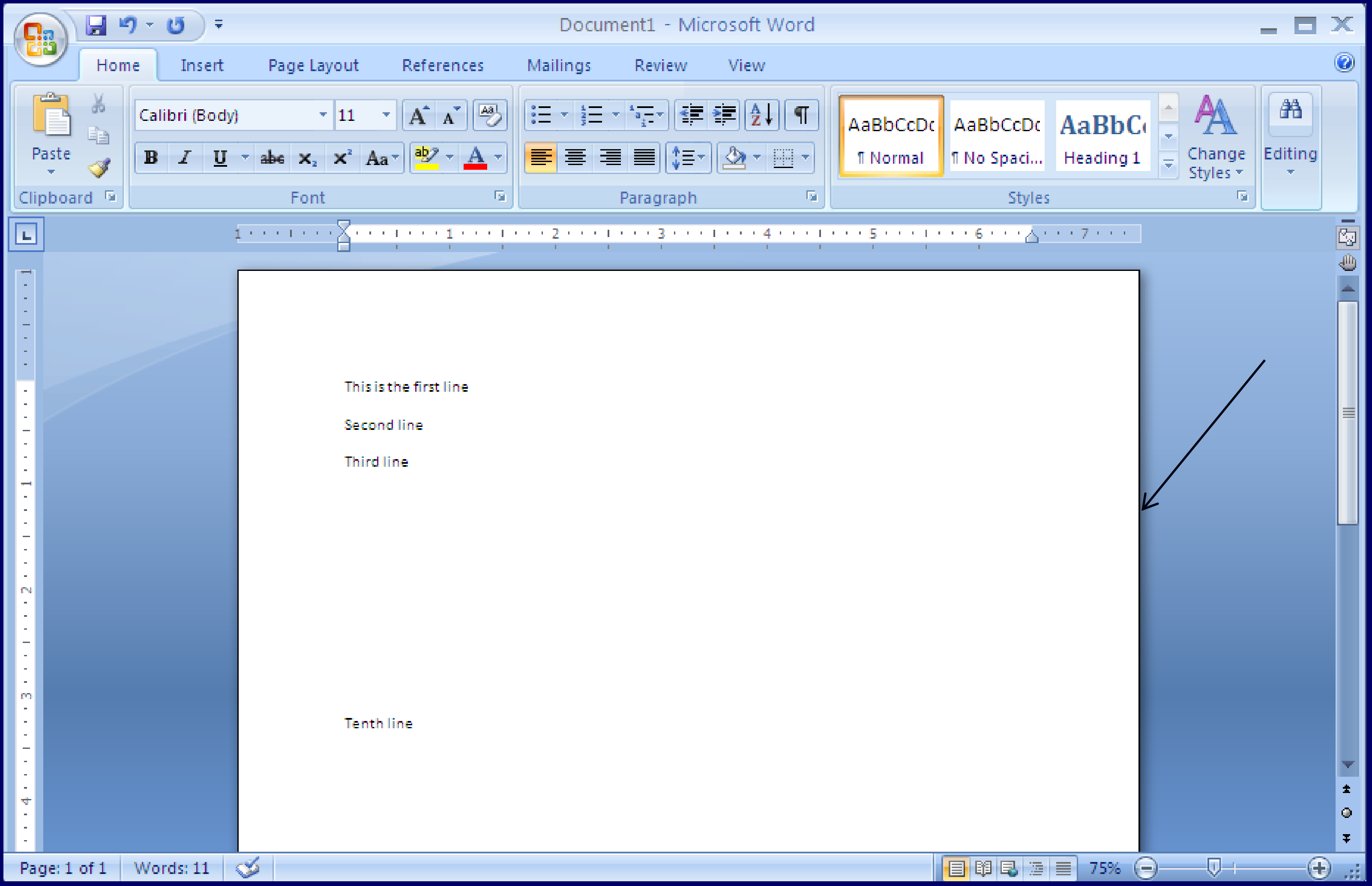
Which data structure should be used here?



Which data structure should be used here?



Which data structure should be used here?



This Week’s To Do List

- Go through lecture slides – make sure you try the code snippets
- Try the lecture’s programs posted on course website