Final Review

CSE1030 – Introduction to
Computer Science II

Reviewing for the Final

- Lab Tests:
  - Sect 01: **Tues** Nov 27
  - Sect 02: **Thurs** Nov 29
- Final: **Thursday December 6, 2:00 pm**

Don’t forget to…

- Forget about writing code
  - Focus on the "Big Ideas" and theory
- "Think Like A Prof" – How would you test whether somebody knows something?
- Review the Lecture notes
- Review the Readings
  - Textbook
  - Course Notes

CSE1030 – Lecture #12

- Introduction
- Java GUI Programming
- Sequential versus Event-Driven
- We’re Done!

Textual (Console) Interfaces

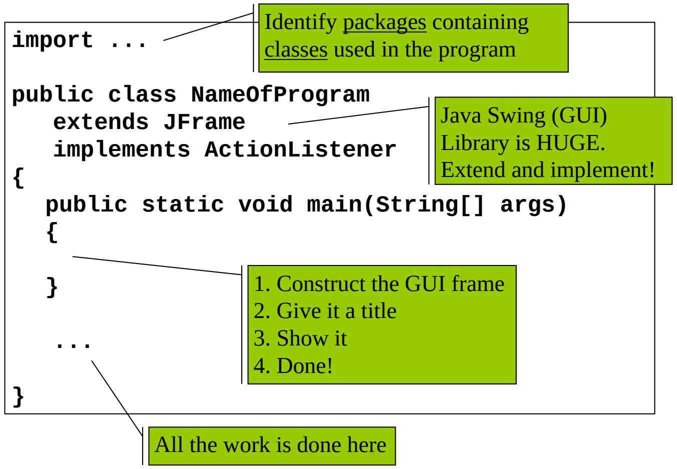
- Older Interaction Style
- Provides a means to express commands to a computer directly via typing and reading text
- May use function keys, single characters, abbreviations, or whole-word commands
- Primarily used today for older applications (e.g., ftp, telnet, Unix command-line)
- Can be difficult for Novices
- Often preferred by Expert users

Graphical User Interfaces

- Newer Style of Interaction
- Usually involves a Pointing Device and Graphical Display
- Richer Output (Graphics, Sound, Video)
- Several Variations
  - Point & Click (web pages)
  - Question & Answer (MS Windows “Wizards”)
  - Forms (Data Entry, Spread Sheets)
  - WIMP (Windows, Icons, Menus, Pointers)
- Can be easier for Novices
- May not be preferred by Experts

GUI Programing with Java

- GUI Programming is accomplished with the javax.swing package
- Sun’s Swing toolkit is Java’s most advanced toolkit, and largest API
- Before Swing…
  - AWT (abstract windowing toolkit)
  - Most of AWT is now obsolete…
  - but AWT still used for a few things (drawing, images, etc.)
- Swing still uses many features of AWT



```
import ...

public class NameOfProgram
   extends JFrame
   implements ActionListener
{
  public static void main(String[] args)
  {

  }

   ...

}
```

Identify packages containing classes used in the program

Java Swing (GUI) Library is HUGE. Extend and implement!

1. Construct the GUI frame
2. Give it a title
3. Show it
4. Done!

All the work is done here

JFrame

- Java GUI Programs are instances of JFrame
  - JFrame is extended to make our own class
- Interaction is received through listeners
  - Listeners are implemented interfaces
  - There are listeners for many different kinds of input (keyboard, mouse, windows opening or closing, and many more)
- So we must be comfortable with extending classes and implementing interfaces

Sequential Programming

- In sequential programs, the program is in control
- The user is required to synchronize with the program:
  - Program tells user it’s ready for more input
  - User enters more input and it is processed
- Examples:
  - Command-line prompts (DOS, UNIX)
  - Command-line programs (ftp, telnet)

Event-driven Programming

- Instead of a user synchronizing with the program, the program synchronizes with, or reacts to, the user
- All communication from user to computer occurs via *events* and the code that handles the events
- An event is an action that happens in the system, such as:
  - A mouse button pressed or released
  - A key-press on the keyboard
  - A window is moved, resized, closed, etc.

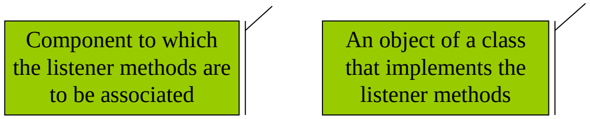
Classes of Events

- Typically two different classes of events:
  - User-initiated events
    - Events that result directly from a user action (e.g., mouse click, move mouse, key press)
  - System-initiated events
    - Events created by the system, as it responds to user action (e.g., scrolling text, re-drawing a window)
- Both classes of events need to be processed
- User-initiated events may generate system-generated events

Installing Listeners

- It is not enough simply to implement the methods of a listener
- The listener must also be installed (or “added”)
- Furthermore, it must be installed for the component to which the listener methods are to be associated
- Thus (from our example program)

```
enterField.addKeyListener(this);
```



Component to which the listener methods are to be associated

An object of a class that implements the listener methods

Installing Listeners (2)

- Signature for the addKeyListener method:
  `public void addKeyListener(KeyListener)`
- Description:
  - Adds the specified key listener to receive key events from this component.
- In our example, we used this as the “specified key listener”
  - Indeed, the current instance of our extended JFrame class (“this”) is a key listener because it implements the key listener methods
- Result: when a key-press event occurs on the enterField component, the keyPressed method in our extended JFrame class will execute!

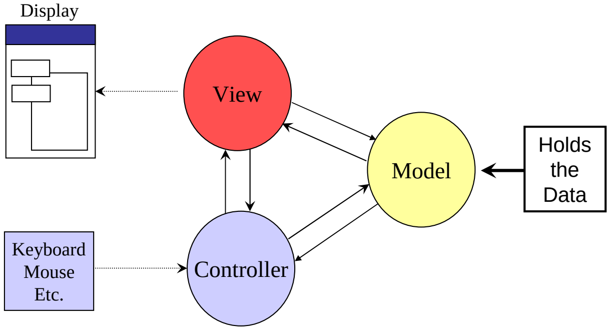
Adapter Classes

- What is an adapter class?
  - A class provided as a convenience in the Java API
  - An adapter class includes an empty implementation of the methods in a listener
  - Programmers extend the adapter class and implement the methods of interest, while ignoring methods of no interest

CSE1030 – Lecture #13

- Review
- MVC
- Game Programming
- We’re Done!

MVC Schematic



Controller Tasks

- Receive user inputs from mouse and keyboard
- Map these into commands that are sent to the model and/or viewport to effect changes in the view
- E.g., detect that a button has been pressed and inform the model that the button stated has changed

Model Tasks

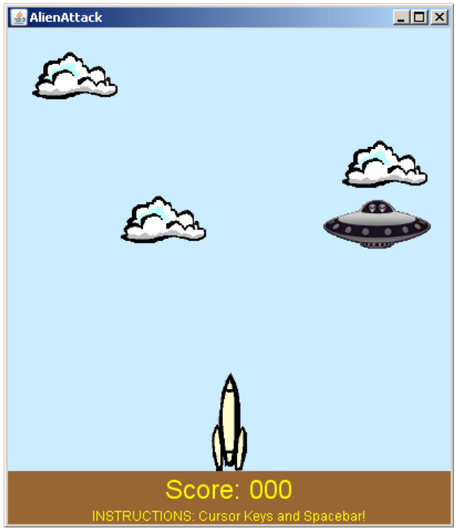
- Store and manage data elements, such as state information
- Respond to queries about its state
- Respond to instructions to change its state
- E.g., the model for a button can be queried to determine if the button is pressed

View tasks

- Implements a visual display of the model
- E.g., a button has a coloured background, appears in a raised perspective, and contains an icon and text; the text is rendered in a certain font in a certain colour

Benefits of MVC Architecture

- Improved maintainability
  - Due to modularity of software components
- Promotes code reuse
  - Due to OO approach (e.g., subclassing, inheritance)
- Model independence
  - Designers can enhance and/or optimize model without changing the view or controller
- Plug-able look and feel
  - New L&F without changing model
  - Multiple views use the same data



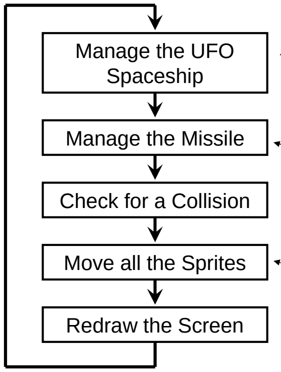
Score: 000

INSTRUCTIONS: Cursor Keys and Spacebar!

AlienAttack

- Controllable Missile
- Randomised UFO spaceship
- Animated Explosion
- Points (Score)

The Main Loop

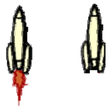


1. Wait for random delay
2. Start new screen transit, with random: side, height, speed
3. Need special handling of the explosion

If the missile is moving, wait for it to hit the top of the screen, and then stop it

Need a way to track all of the sprites, with their speeds and directions

Media

- This is the most important part of the game:
  -
  -
  - 

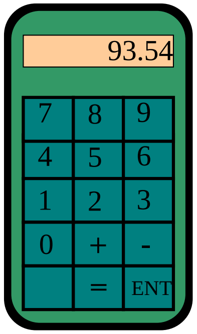
CSE1030 – Lecture #14

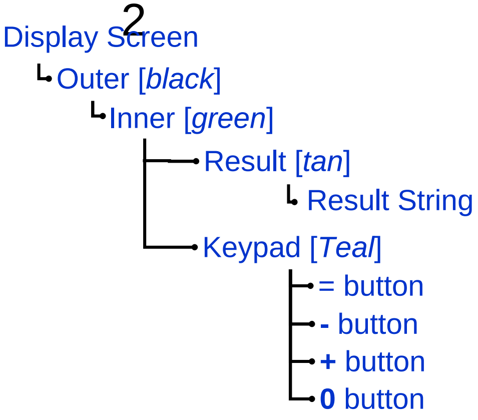
- Review
- Containment Hierarchy
- Component Layout
- We’re Done!

Containment Hierarchy

- A window is made up of a number of nested interactive objects (e.g., buttons, text fields, other windows)
- Relationship of objects is expressed by a *containment hierarchy* (a.k.a. *interactor tree*)
  - based on screen geometry of objects
  - represents the hierarchy / nesting of the objects

Containment Hierarchy - Example 2



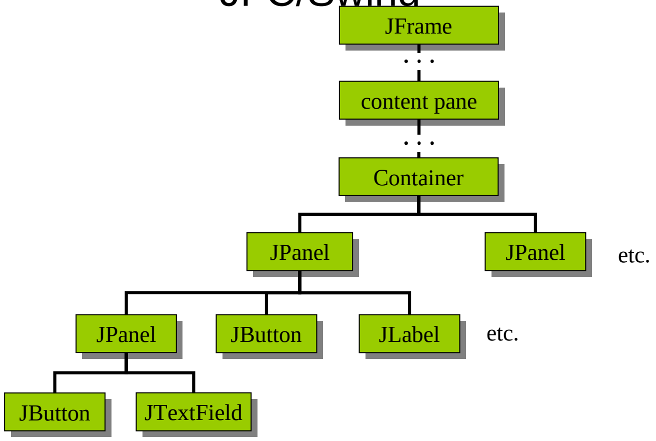


Display Screen
- Outer [*black*]
  - Inner [*green*]
    - Result [*tan*]
      - Result String
    - Keypad [*Teal*]
      - **=** button
      - **-** button
      - **+** button
      - **0** button

Containers

- Components are placed in *containers*
- A JFrame is a *top-level container*
  - It exists mainly as a place for other components to paint themselves
  - Cannot place a JFrame inside a JFrame
- A JPanel is an *intermediate container*
  - Sole purpose is to simplify the positioning of interactive objects, such as buttons or text fields
  - Other intermediate containers are *scroll panes* (JScrollPane) and *tabbed panes* (JTabbedPane)
  - Can place a JPanel inside a JPanel (or inside a JFrame, via the content pane)

Containment Hierarchy for JFC/Swing



JFrame

. . .

content pane

. . .

Container

JPanel

JPanel

etc.

JPanel

JButton

JLabel

etc.

JButton

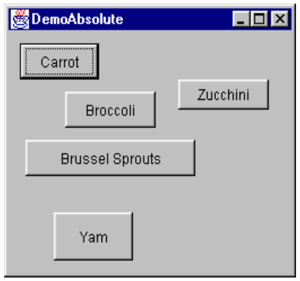
JTextField

Absolute Positioning

- Component position and size explicitly specified…
  - X and Y screen coordinates
  - Width and height of component
  - Units: pixels (typically)

Example Program

DemoAbsolute.java



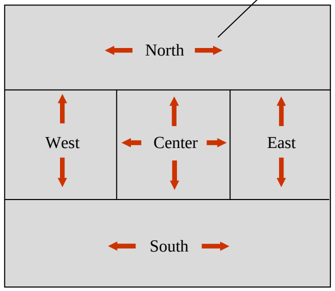
BorderLayout

- Places components in one of five regions
  - North, South, East, West, Center
- Support for struts and springs
  - Struts (✓)
    - Can specify ‘hgap’, ‘vgap’
  - Springs (×)
    - Inter-component space is fixed
  - Components expand to fill space in region

Border Layout (2)

- Components ‘expand’ (or ‘stretch’) to fill space as follows

Expand direction



FlowLayout

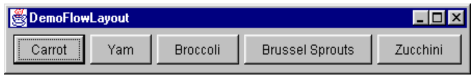
- Arranges components in a group, left-to-right
  Wraps components to new line if necessary
- Support for struts and springs
  - Struts (✓)
    - Can specify ‘hgap’, ‘vgap’
  - Springs (×)
    - Inter-component space is fixed
  - Component size is fixed
- Space is added before / after / below the entire group of components to fill available space

Example Progra

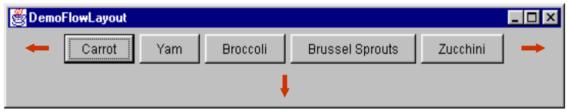
Default for FlowLayout…
struts : hgap = vgap = 5,
alignment = center

```
Invocation: java DemoFlowLayout 5 c
```

Launch



Resize



Fill available space before/after/below group of components

GridLayout

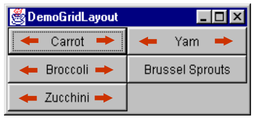
- Arranges components in a rectangular grid
  The grid contains equal-size rectangles
- Support for struts and springs
  - Struts (✓)
    - Can specify ‘hgap’, ‘vgap’
  - Springs (×)
    - Inter-component space is fixed
  - Components expand to fill rectangle

Example Program (2)

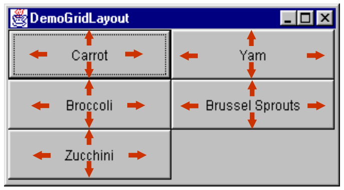
No struts

```
Invocation: java DemoGridLayout 0
```

Launch



Resize



Equal-size rectangles

BoxLayout

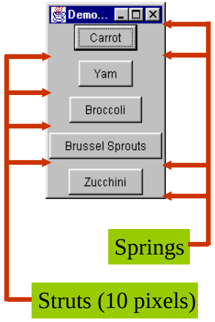
- Arranges components vertically or horizontally Components do not wrap
- Support for struts and springs
  - Struts (✓)
    - Can specify ‘rigid areas’
  - Springs (✓)
    - Can specify ‘horizontal glue’ or ‘vertical glue’
  - Components expand if maximum size property is set

Example Program (3)

Enable struts and springs demo

```
Invocation: java DemoBoxLayout c e
```

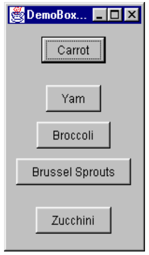
Launch



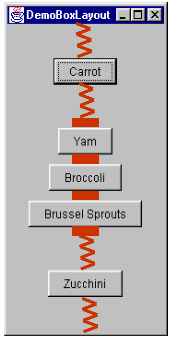
Springs

Struts (10 pixels)

Resize



Resize more



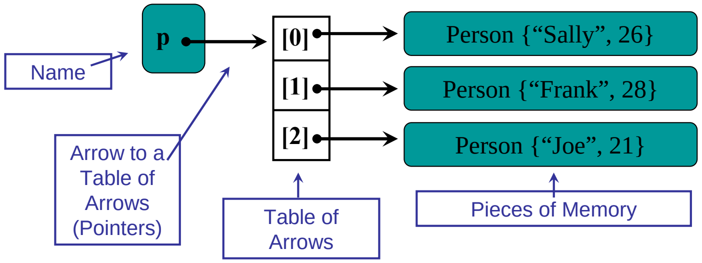
CSE1030 – Lecture #15

- Introduction to Arrays
- Constant Arrays Examples
- Dynamic Arrays
- We’re Done!

New Idea: An Array is…

- A Name, and a **Table of Arrows (Pointers)**, to Blocks of Memory:

```
Person[] p = new Person[] {
   new Person("Sally", 26),
   new Person("Frank", 28),
   new Person("Joe", 21),
};
```



p

[0]

[1]

[2]

Person {“Sally”, 26}

Person {“Frank”, 28}

Person {“Joe”, 21}

Name

Arrow to a Table of Arrows (Pointers)

Table of Arrows

Pieces of Memory

Days of the Week – Better Solution

```
import java.util.*;

class example2b
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.println("Enter a 'Day of the Week' "
                            +"(1 <= integer <= 7):");

      int day_of_week = 0;
      try {
         day_of_week = in.nextInt();
      }
      catch(Exception e)
      {
         day_of_week = 0;
      }
```

```
      final String[] days = {
         "Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday", "Sunday",
      };

      if(day_of_week < 1 || day_of_week > 7)
         System.out.println("Bad Input - try again");
      else
         System.out.println( days[day_of_week - 1] );
   }
}
```

Array Summary (1/2)

- Declare Arrays:

```
int[] someNumbers;
String[] words;
```

- Constructing Empty Arrays:

```
someNumbers = new int[10];
String[] words = new String[3];
```

- Initialising with Hardcoded Values:

```
int[] somenumbers = {
   2, 3, 5, 7, 11,
};
String[] words = {
   "Hello", "Good Bye"
};
```

Array Summary (2/2)

- Using Arrays:

```
String s = words[2];
int n = someNumbers[i];

somenumbers[3] = 17;
```

(Note: the Index must be an `int`)

- Array Size:

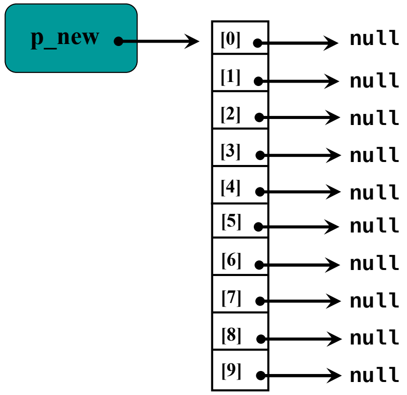
```
someNumbers.length
words.length
```

Resize – Larger

1. Need to create a new Larger array
2. Copy the objects over to the new array
3. Switch over to the new array

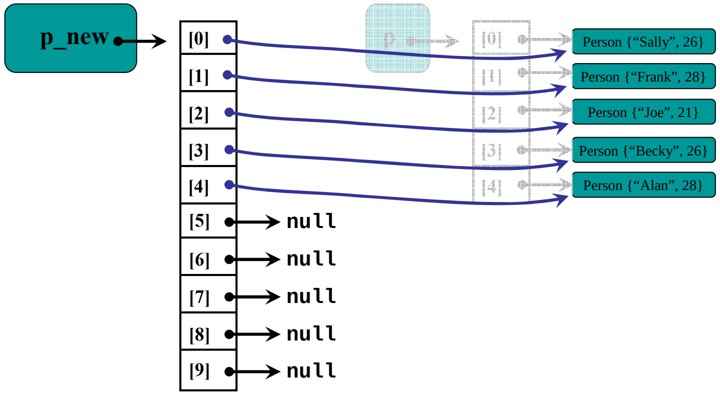
1 – Create a New Larger Array

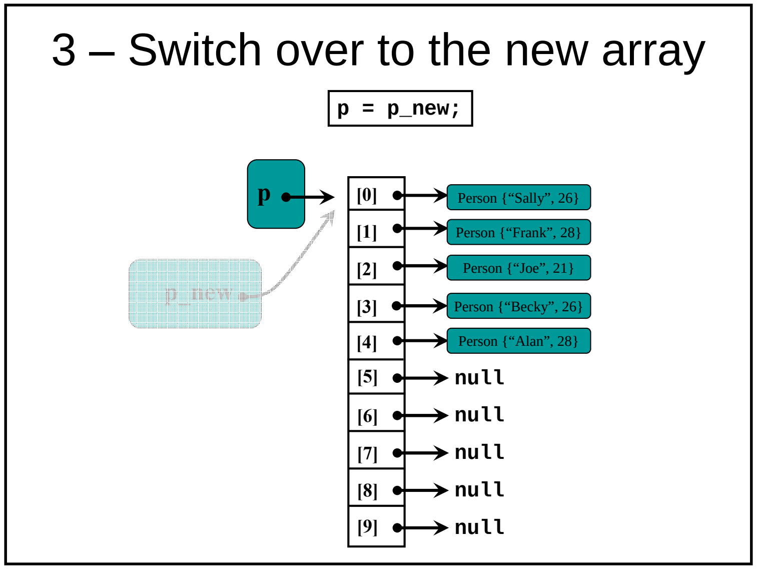
```
Person[] p_new = new Person[p.length + 5];
```



2 – Copy the objects over to the new array

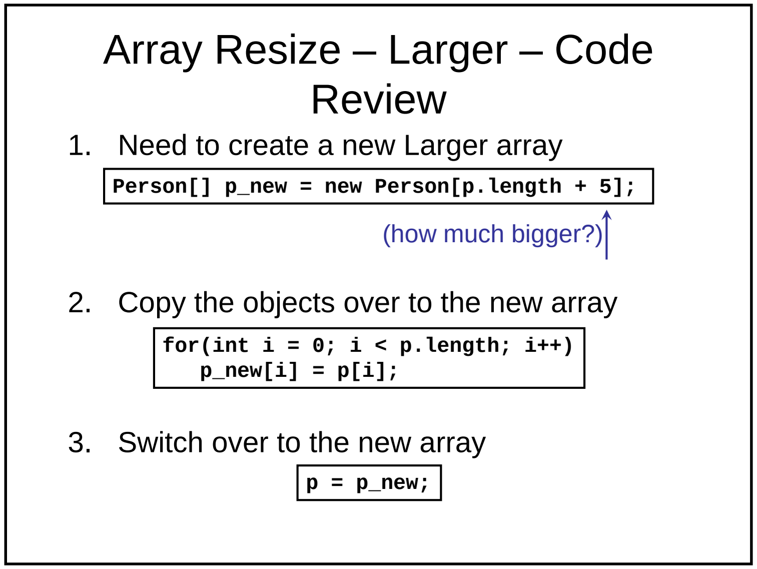
```
for(int i = 0; i < p.length; i++)
   p_new[i] = p[i];
```





3 – Switch over to the new array

```
p = p_new;
```



Array Resize – Larger – Code Review

1. Need to create a new Larger array

```
Person[] p_new = new Person[p.length + 5];
```

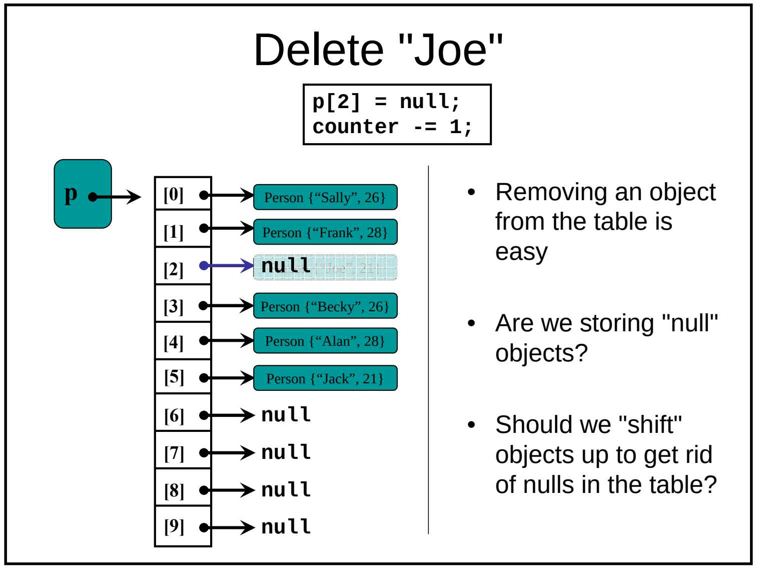
(how much bigger?)

2. Copy the objects over to the new array

```
for(int i = 0; i < p.length; i++)
   p_new[i] = p[i];
```

3. Switch over to the new array

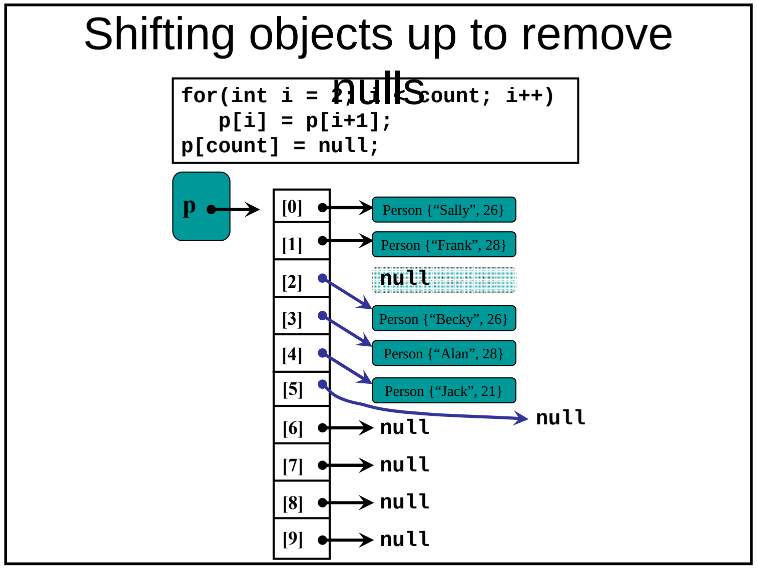
```
p = p_new;
```



Delete "Joe"

```
p[2] = null;
counter -= 1;
```

- Removing an object from the table is easy
- Are we storing "null" objects?
- Should we "shift" objects up to get rid of nulls in the table?



Shifting objects up to remove nulls

```
for(int i = 2; i < count; i++)
   p[i] = p[i+1];
p[count] = null;
```

Dynamic Arrays – Summary

- Three important Operations:
  - – Adding } These are tricky, because we may have to resize the array, or shift objects around – Inefficient!
  - – Deleting
  - – Iterating } Fast, Fast, Fast! It's hard to beat **`p[i]`**
- Iterating is easy – just access what you want

```
for(int i = 0; i < count; i++)
   System.out.println(p[i]);
```

Arrays – The Big Questions

- Do we even allow the expensive operations (adding or removing to/from the middle of the list)
- Do we leave "null" values, or shift to remove them?
- Space / Time Trade-off:
  - – How big an array do we start with?
  - – By how many slots do we enlarge the array?
  - – Do we ever shrink an array? By how much?

Efficient Operations

- Adding or Deleting to/from the End of the Array is fast (no Shifting), so those are safe operations to support, and to use
- Adding or Deleting to/from the Middle of the Array may require Shifting, which is inefficient
  - – So maybe these operations should not be allowed?
  - – Or only infrequently used?
  - – Or they should be grouped together so all the shifting can be done at once?
  - – Or we can leave "nulls", but that complicates things

Shifting Objects Up/Down

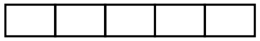
- There is a Trade-off:
  - – Shifting to Remove null values:
    - Slower, but more memory efficient
    - Makes it easy to insert (**`count`** = available slot)
    - May not be so bad, if there are only a few deletes
  - – Leaving nulls:
    - Means the user can't store nulls in the array
    - Faster Deletion
    - not necessarily faster Addition
      (searching for nulls is time consuming, changes order)
    - Can waste a Huge amount of memory
- In the end, it depends upon the properties of your data, and the requirements of your application. Experiment!!

CSE1030 – Lecture #16

- Review: Arrays
- Regular 2D Arrays
- Irregular 2D Arrays
- We're Done!

The Big Idea so far…

- When data "looks like" this:



(and you can't use, or don't need the complexity of, a Collection)

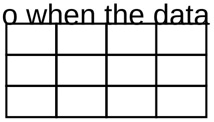
- Use an array:

```
Object[] array = new Object[5];
```

array =	array[0]
	array[1]
	array[2]
	array[3]
	array[4]

New Idea… What about Tables?

- What do we do when the data "looks like" this?



- Use a 2-Dimensional array:

```
Object[3][4] array = new Object[3][4];
```

array =	array[0][0]	array[0][1]	array[0][2]	array[0][3]
	array[1][0]	array[1][1]	array[1][2]	array[1][3]
	array[2][0]	array[2][1]	array[2][2]	array[2][3]

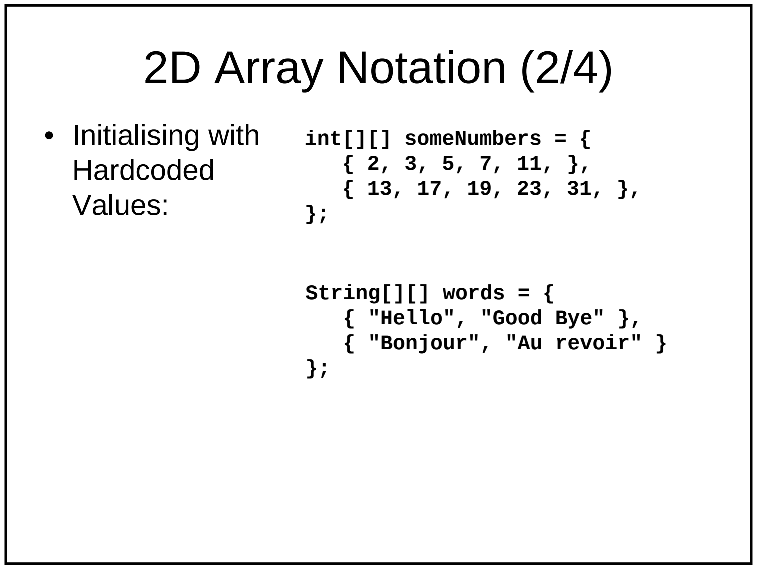
2D Array Notation (1/4)

- Declare Arrays:

```
int[][] someNumbers;
String[][] words;
```

- Constructing Empty Arrays:

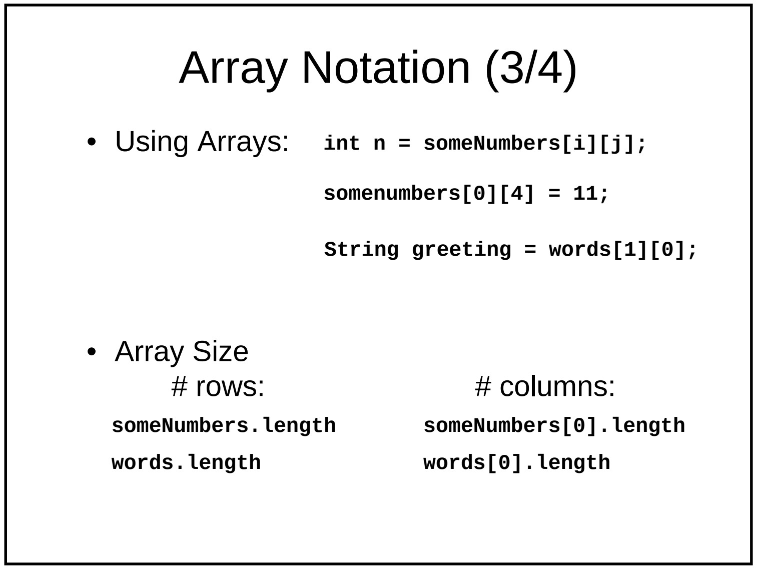
```
someNumbers = new int[10][5];
String[] words = new String[3][2];
```



2D Array Notation (2/4)

- Initialising with Hardcoded Values:

```
int[][] someNumbers = {
   { 2, 3, 5, 7, 11, },
   { 13, 17, 19, 23, 31, },
};


String[][] words = {
   { "Hello", "Good Bye" },
   { "Bonjour", "Au revoir" }
};
```



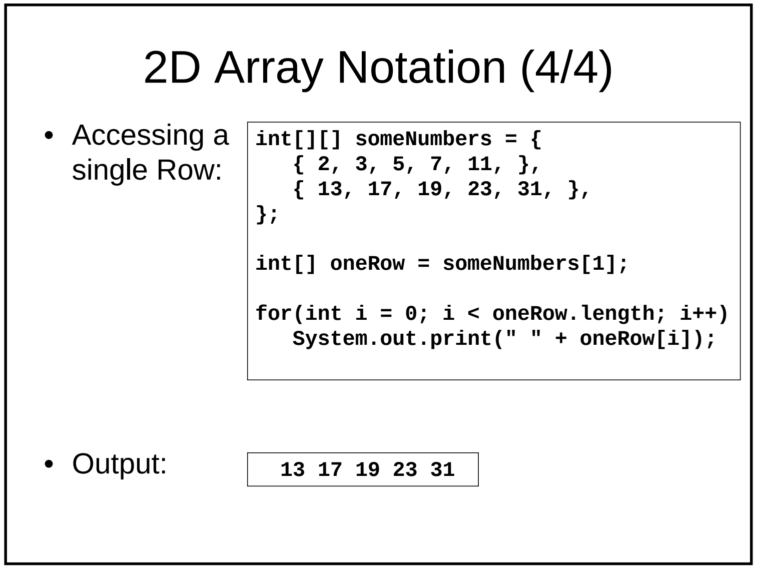
Array Notation (3/4)

- Using Arrays:

```
int n = someNumbers[i][j];

somenumbers[0][4] = 11;

String greeting = words[1][0];
```

- Array Size

# rows:	# columns:
`someNumbers.length`	`someNumbers[0].length`
`words.length`	`words[0].length`



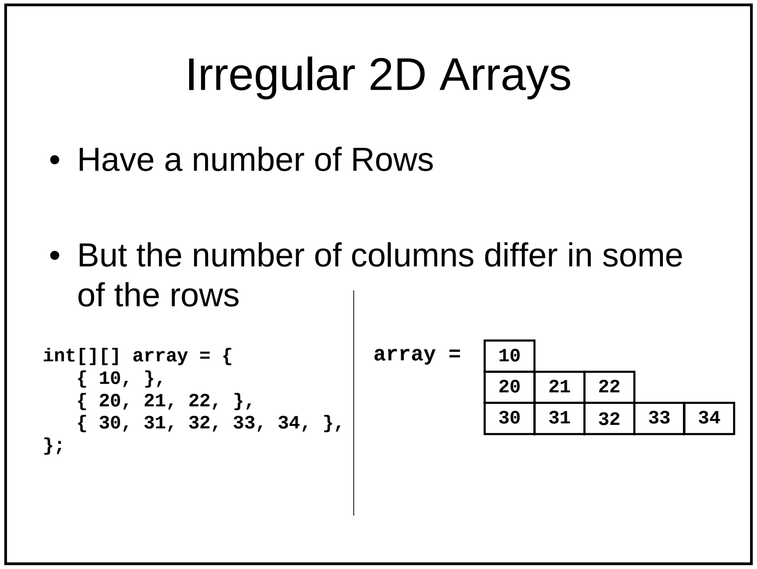
2D Array Notation (4/4)

- Accessing a single Row:

```
int[][] someNumbers = {
   { 2, 3, 5, 7, 11, },
   { 13, 17, 19, 23, 31, },
};

int[] oneRow = someNumbers[1];

for(int i = 0; i < oneRow.length; i++)
   System.out.print(" " + oneRow[i]);
```

- Output:

```
 13 17 19 23 31
```



Irregular 2D Arrays

- Have a number of Rows

- But the number of columns differ in some of the rows

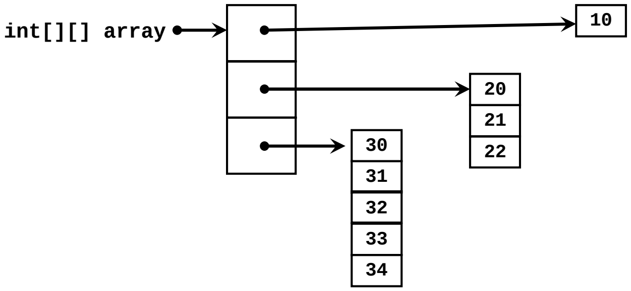
```
int[][] array = {
   { 10, },
   { 20, 21, 22, },
   { 30, 31, 32, 33, 34, },
};
```

array =

10				
20	21	22		
30	31	32	33	34

How are Irregular 2D Arrays Possible?

- A 2D Array is really an "Array of Arrays":



Advanced Usage of Arrays…

- You can have higher-dimensional arrays:

```
int[][][] array = {
         ...
};
```

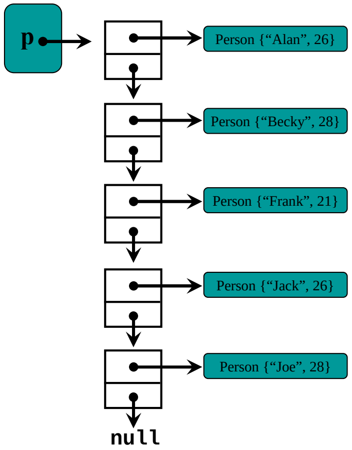
- You can have arrays of Objects:

```
Object[] array = {
    new Moons(),
    new ChessBoard(),
    new Integer(10),
    new String[] { "Hi", "Bye" },
};
```

CSE1030 – Lecture #17

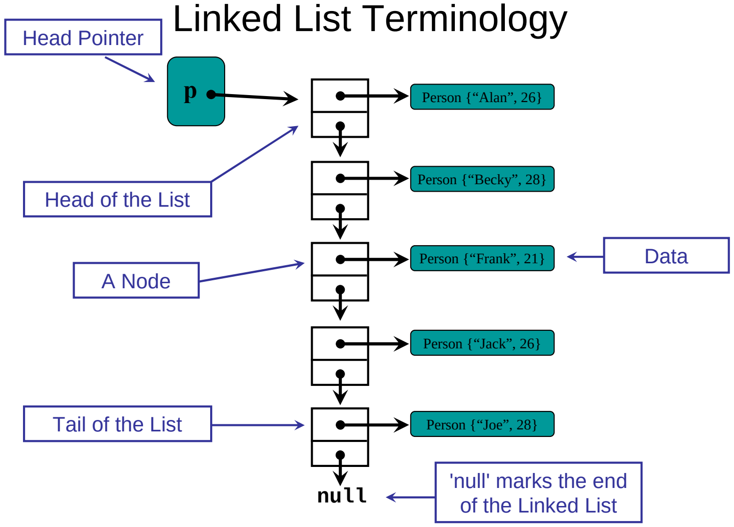
- Review
- Introduction to Linked Lists
- We're Done!

How about we use lots of Little Blocks of Memory, instead of 1 Big one?



- Each Little Block holds an arrow (reference, pointer) to the data
- Each Little Block also has to provide a way to find the next little block

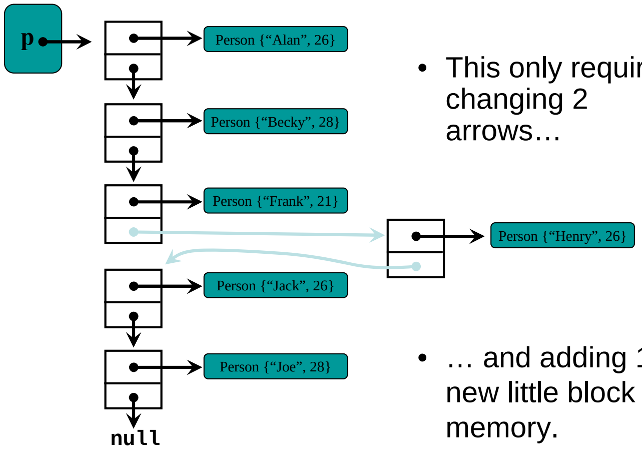
Linked List Terminology



Implications

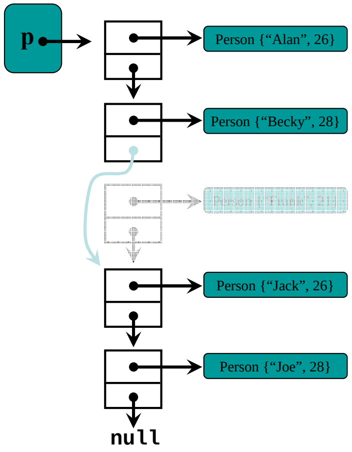
- We are still storing a collection of arrows (or "references", or "pointers") as we did when we used arrays
- But because the arrows are in their own individual little pieces of memory, nothing has to be shifted to insert new ones
- There are other benefits too…

Now, Inserting Henry is Easy!



- This only requires changing 2 arrows…
- … and adding 1 new little block of memory.

Deletion is also Easy



- Deleting "Frank" only requires us to update 1 pointer – Fast!

Arrays versus Linked Lists

Arrays

- Good:
  - Access to any element is very fast: `p[i]`
  - Adding / Deleting from the **End** is **Fastest** (but can cause Resizing)
  - Efficient on Memory (only 1 arrow per Data item)
  - But empty slots waste memory
- Bad:
  - Insertion / Deletion anywhere but the end of the array
  - Resizing

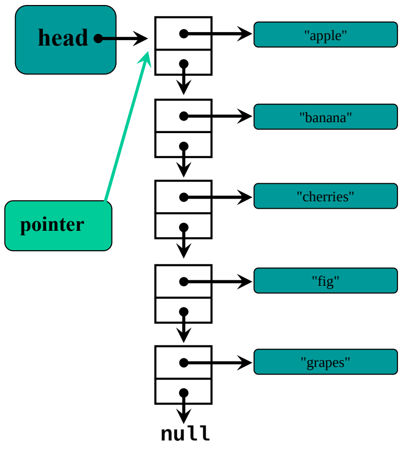
Linked Lists

- Good:
  - Insertion / Deletion is easy (just update some arrows)
  - Insertion or Deletion at the **Top** of the list is **Fastest**
  - There is no "Resizing Cost"
- Bad:
  - Accessing an element requires us to iterate along the List – Slower than array
  - Wastes more Memory (2 arrows per Data item)
  - Although it, doesn't have empty slots

CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Linked List Iteration



- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

```
// now we want to output the list:
Node pointer = head;

int i;
while(pointer != null)
{
   System.out.println("  " + i++ + " " + pointer.data);

   pointer = pointer.next;
}

System.out.println("Done!");
```

Start at the head ("top") of the list

Use the Data

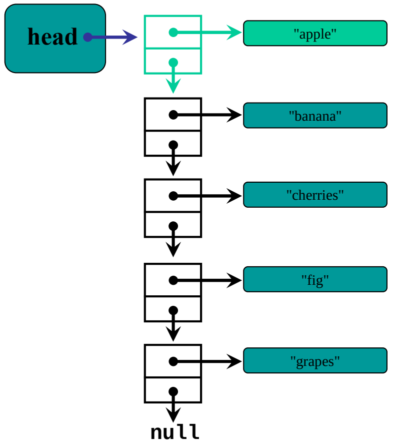
Move the pointer on down the list

Inserting Nodes into a Linked-List

- Insertion requires us to create a new Node, and update a pointer
- There are three cases:

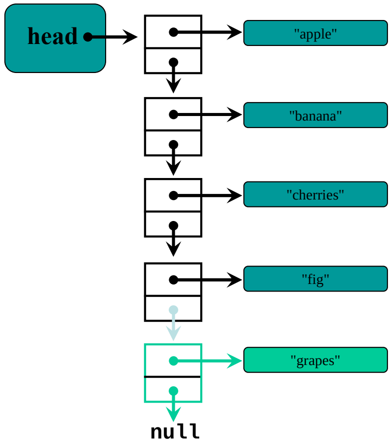
1. Inserting at the **head** of the list
2. Inserting at the **end** of the list
3. Inserting in the **middle**

Inserting at the Beginning



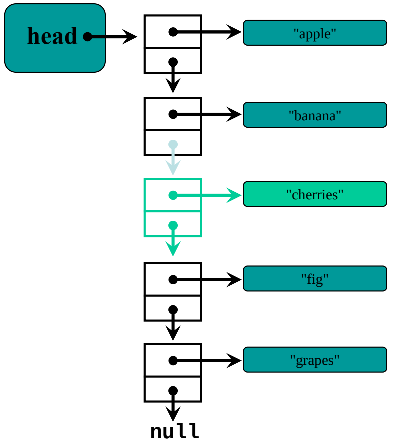
- Next we update the head pointer and we're done
- Let's look at the code…

Inserting at the End



- To insert at the end of the list we have to change the 'next' pointer of the last node…
- and we have to add a new node with a 'next' pointer that is null.
- Let's look at the code

Inserting in the Middle

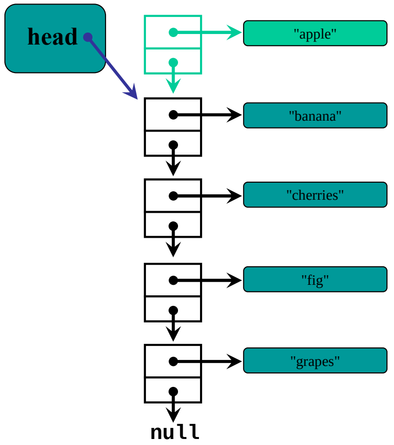


- To insert in the middle of the list we have to find the node **above** where the new node should go…
- because that's the node where the 'next' pointer has to be changed.

Deleting Nodes from a Linked-List

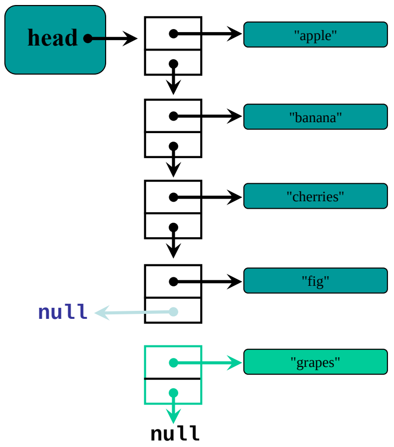
- Deletion only requires us to update a pointer
- There are three cases:
  1. Deleting from the **head** of the list
  2. Deleting from the **end** of the list
  3. Deleting from the **middle**

Deleting from the Beginning



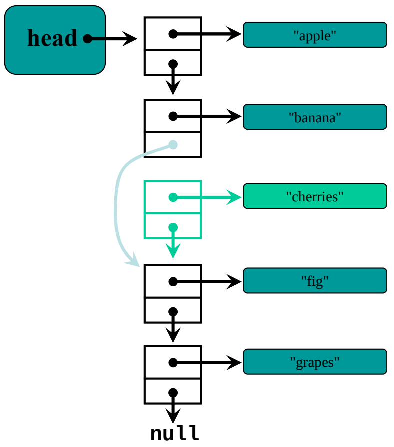
- Here we move the head pointer one node down the list…

Deleting from the End



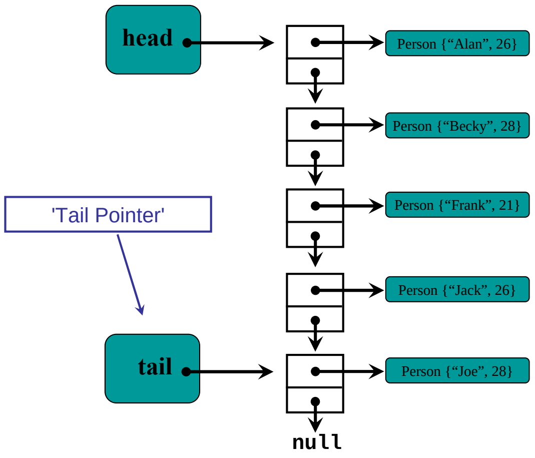
- To delete from the end of the list we have to change the 'next' pointer of the **second-last** node to null…

Deleting from the Middle



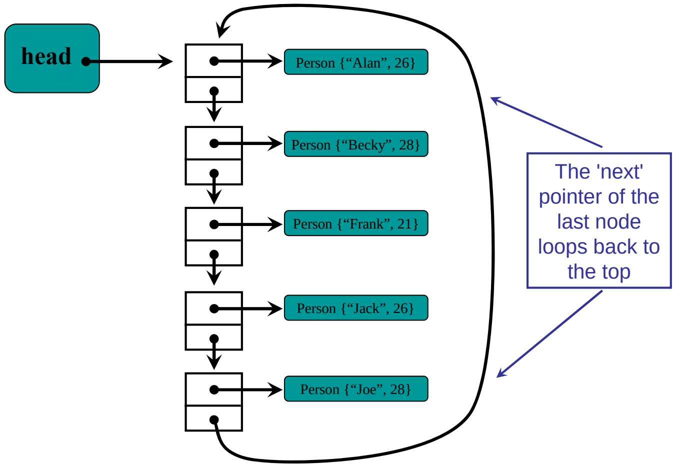
- Once we have updated the preceding 'next' pointer, the skipped node has been removed from the list

Linked List with Tail Pointer



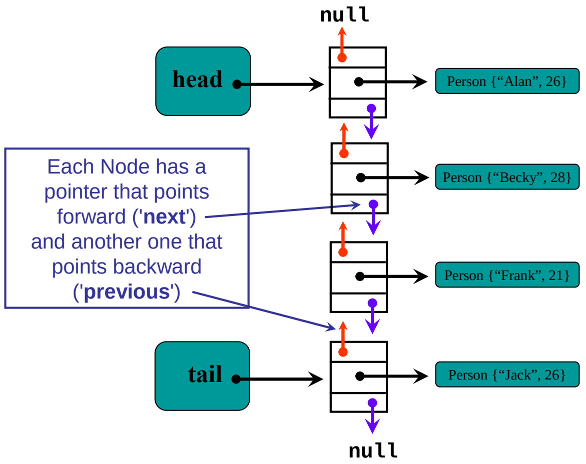
'Tail Pointer'

Circular Linked Lists



The 'next' pointer of the last node loops back to the top

Doubly Linked-List

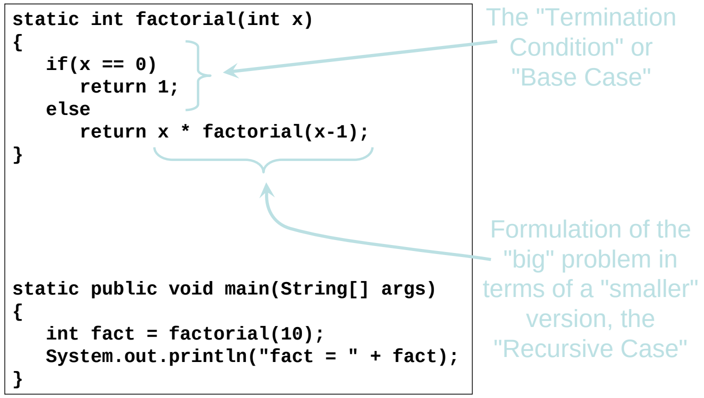


Each Node has a pointer that points forward ('**next**') and another one that points backward ('**previous**')

CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

Let's Re-examine the Code

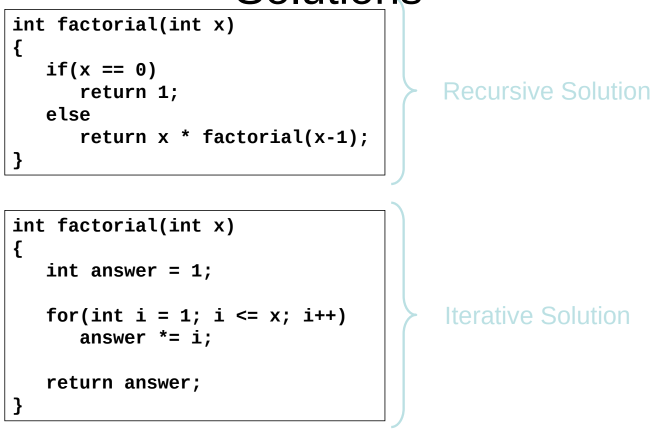


```
static int factorial(int x)
{
   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}


static public void main(String[] args)
{
   int fact = factorial(10);
   System.out.println("fact = " + fact);
}
```

The "Termination Condition" or "Base Case"

Formulation of the "big" problem in terms of a "smaller" version, the "Recursive Case"

Iterative versus Recursive Solutions



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

Recursive Solution

```
int factorial(int x)
{
   int answer = 1;

   for(int i = 1; i <= x; i++)
      answer *= i;

   return answer;
}
```

Iterative Solution

How do the Result values get Returned?

- Same functionality, more print statements…

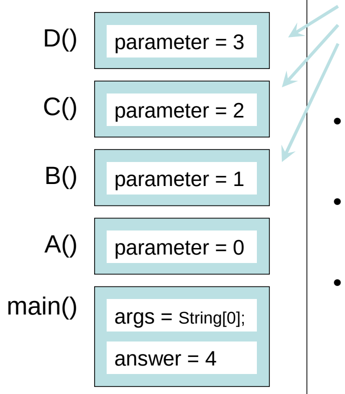
```
static int factorial(int x)
{
   System.out.println("factorial(" + x + ") called!");

   if(x == 0)
   {
      System.out.println("factorial(0) returned: 1");
      return 1;
   }
   else
   {
      int retval =  x * factorial(x-1);
      System.out.println("factorial(" + x + ")"
                         + " returned: " + retval);
      return retval;
   }
}
```

Output of Improved Version

```
>java factorialRecursiveVerbose
factorial(10) called!
factorial(9) called!
factorial(8) called!
factorial(7) called!
factorial(6) called!
factorial(5) called!
factorial(4) called!
factorial(3) called!
factorial(2) called!
factorial(1) called!
factorial(0) called!
factorial(0) returned: 1
factorial(1) returned: 1
factorial(2) returned: 2
factorial(3) returned: 6
factorial(4) returned: 24
factorial(5) returned: 120
factorial(6) returned: 720
factorial(7) returned: 5040
factorial(8) returned: 40320
factorial(9) returned: 362880
factorial(10) returned: 3628800
fact = 3628800
```

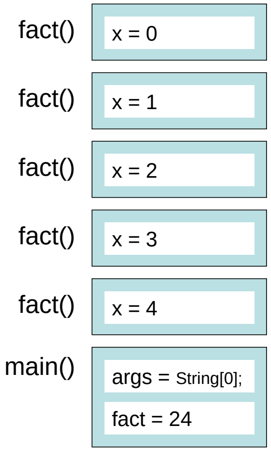
- Same functionality, more print statements…
- Shows both:
  - The **Calls** recursing down to the terminating case
  - And the **Returns** recursing back out to the answer

Execution Stack


D() parameter = 3
C() parameter = 2
B() parameter = 1
A() parameter = 0
main() args = String[0]; answer = 4

- Every time that Java starts a new function, it creates a "stack frame", a unique place to hold the local variables for that function
- When a function returns, the stack frame goes away
- This is called the "execution stack"
- Consequently, in this example each function has their own distinct variable called "parameter"

Recursive Execution Stack Example

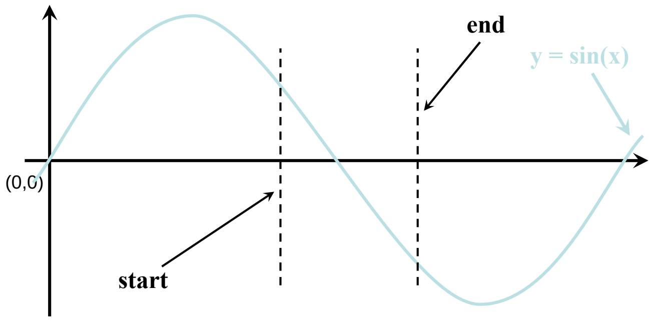

fact() x = 0
fact() x = 1
fact() x = 2
fact() x = 3
fact() x = 4
main() args = String[0]; fact = 24

- Every time we recurse, Java creates a new stack frame, within which the variables exist.
- This is how recursion works.

```
int fact(int x)
{
   if(x == 0)
      return 1;
   else
      return x * fact(x-1);
}

public void main(String[] args)
{
   int f = fact(4);
   System.out.println("fact = " + f);
}
```

Mathematical Bisection

- Bisection is a technique used to find the point where a function crosses zero (to find **x** where **f(x) = 0**)
- We sandwich the zero between two points (**start** & **end**)



```
static double bisect(double start, double end)
{
   double mid = (start + end) / 2.0;

   if(Math.abs(start - end) < errorTolerance)
      return mid;

   if(f(mid) > 0)
      return bisect(mid, end);
   else
      return bisect(start, mid);
}
```

Recursive Solution

CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

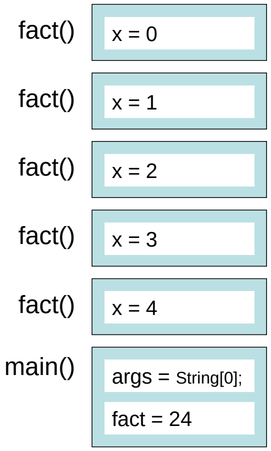
Theory: Definition of Recursion

- A function is **Recursive** if it calls itself (directly or indirectly) from within its own body
- Two components of a Recursive Solution:
  1. A solution to the problem that involves a simpler instance of the problem (called the "**Recursive Case**")
  2. A **Direct Solution** to a simple version of the problem (called the "**Termination Case**", or "**Base Case**")
- Any algorithm can be implemented with either a recursive or iterative algorithm, although some problems are easier to solve one way or the other

Practical: Coding Recursion

- A function is **Recursive** if it calls itself (directly or indirectly) from within its own body
- Recursive Functions always have:
  1. An "if" statement
     - The "if" tests whether the function input is a "**Base Case**"
     - If the input is a Base Case, then a value is returned directly (without calling the function again)
  2. Otherwise, the input requires the "**Recursive Case**"
     - The function calls itself with an argument that is closer to the Base Case than the original argument

How? Recursive Execution Stack



fact() x = 0

fact() x = 1

fact() x = 2

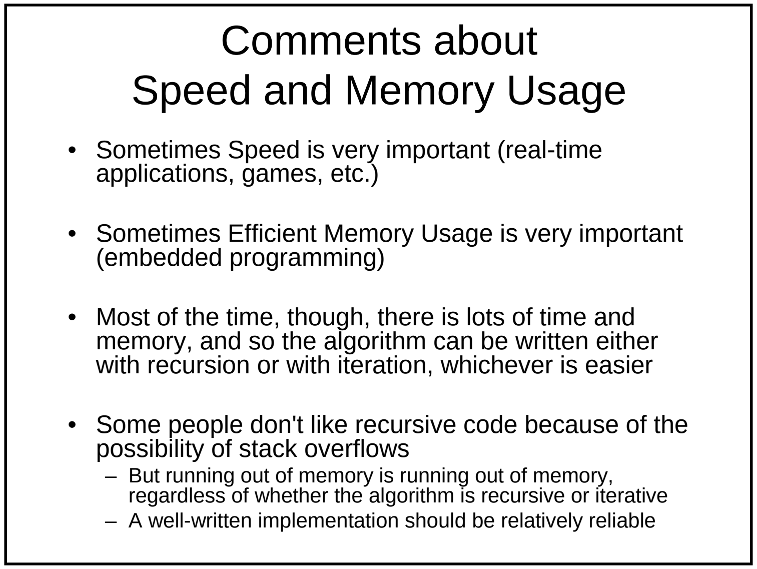
fact() x = 3

fact() x = 4

main() args = String[0]; fact = 24

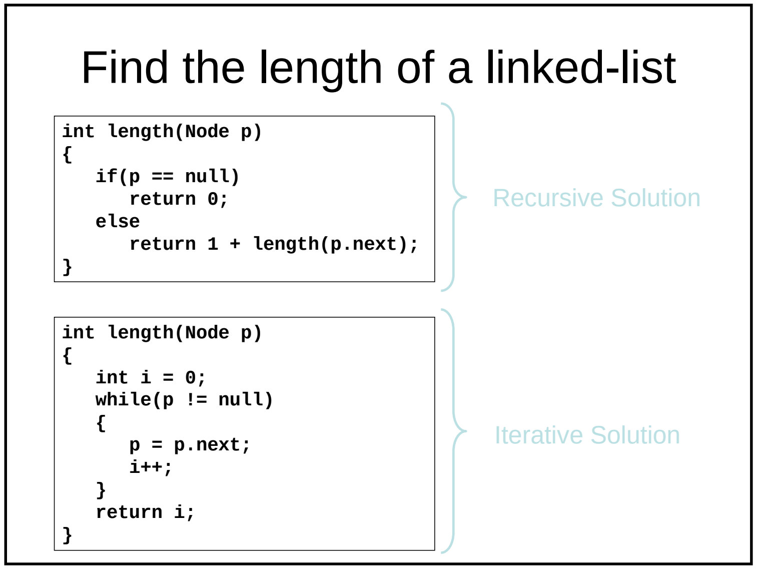
- Every time we recurse, Java creates a new stack frame, within which the variables exist.
- This is how recursion works.

```
int fact(int x)
{
   if(x == 0)
      return 1;
   else
      return x * fact(x-1);
}

public void main(String[] args)
{
   int f = fact(4);
   System.out.println("fact = " + f);
}
```



Comments about Speed and Memory Usage

- Sometimes Speed is very important (real-time applications, games, etc.)
- Sometimes Efficient Memory Usage is very important (embedded programming)
- Most of the time, though, there is lots of time and memory, and so the algorithm can be written either with recursion or with iteration, whichever is easier
- Some people don't like recursive code because of the possibility of stack overflows
  - But running out of memory is running out of memory, regardless of whether the algorithm is recursive or iterative
  - A well-written implementation should be relatively reliable



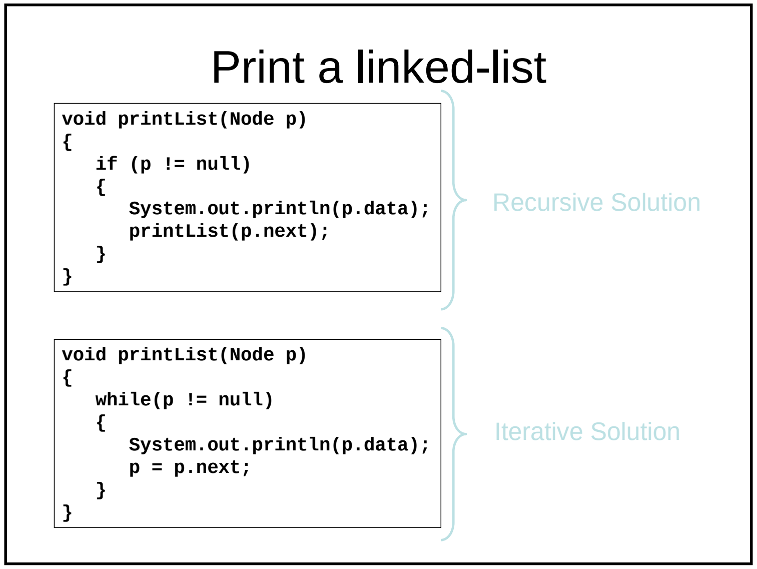
Find the length of a linked-list

```
int length(Node p)
{
   if(p == null)
      return 0;
   else
      return 1 + length(p.next);
}
```

Recursive Solution

```
int length(Node p)
{
   int i = 0;
   while(p != null)
   {
      p = p.next;
      i++;
   }
   return i;
}
```

Iterative Solution



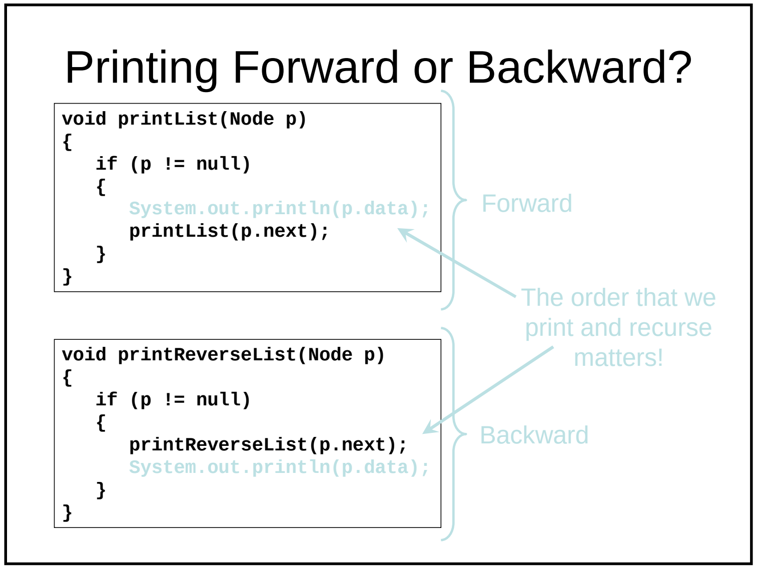
Print a linked-list

```
void printList(Node p)
{
   if (p != null)
   {
      System.out.println(p.data);
      printList(p.next);
   }
}
```

Recursive Solution

```
void printList(Node p)
{
   while(p != null)
   {
      System.out.println(p.data);
      p = p.next;
   }
}
```

Iterative Solution



Printing Forward or Backward?

```
void printList(Node p)
{
   if (p != null)
   {
      System.out.println(p.data);
      printList(p.next);
   }
}
```

Forward

The order that we print and recurse matters!

```
void printReverseList(Node p)
{
   if (p != null)
   {
      printReverseList(p.next);
      System.out.println(p.data);
   }
}
```

Backward

Copying a linked-list

- Copying a linked-list is much easier using recursion…

Recursive Solution:

```
Node copy(Node p)
{
   if(p == null)
      return null;
   else
      return new Node(p.data, copy(p.next));
}
```

Reversing a linked-list

Recursive Solution:

```
Node reverse(Node p)
{
   return reverse(p, null);
}

Node reverse(Node p, Node ancestor)
{
   if(p == null)                  // empty?
      return ancestor;

   Node theNextNode = p.next;     // remember who's next

   p.next = ancestor;             // point this node backwards

   return reverse(theNextNode, p);   // recurse to next node
}
```

Inserting into an Ordered linked-list

Recursive Solution:

```
Node insertInOrder(String key, Node p)
{
   if(p == null || p.data.compareTo(key) >= 0)
      return new Node(key, p);

   else
   {
      p.next = insertInOrder(key, p.next);
      return p;
   }
}
```

Usage:

```
head = insertInOrder("newdata", head);
```

Deleting from an Ordered linked-list

Recursive Solution:

```
Node deleteInOrder(String key, Node p)
{
   if(p == null)
      return p;

   else if (p.data.equals(key))
      return p.next;

   else
   {
      p.next = deleteInOrder(key, p.next);
      return p;
   }
}
```

Usage:

```
head = deleteInOrder("deldata", head);
```

Deleting the last Node of a linked-list

Recursive Solution:

```
Node deleteLast(Node p)
{
   if(p == null || p.next == null)
      return null;

   else
   {
      p.next = deleteLast(p.next);
      return p;
   }
}
```

Usage:

```
head = deleteLast(head);
```

"Two-List" Operations

- All of the Linked-List operations we have seen so far have used only 1 linked-list
- Next, let's look at three operations that combine two linked-lists into one list:
  - Append
  - Shuffle
  - Merge
- For these examples will use the following data:

p = apple → banana → cherries → fig → grapes → null

q = aardvark → bat → cat → dragon → elephant → null

Append

Recursive Solution:

```
Node append(Node p, Node q)
{
   if(p == null)
      return q;

   else
   {
      p.next = append(p.next, q);
      return p;
   }
}
```

Result:

```
apple
banana
cherries
fig
grapes
aardvark
bat
cat
dragon
elephant
```

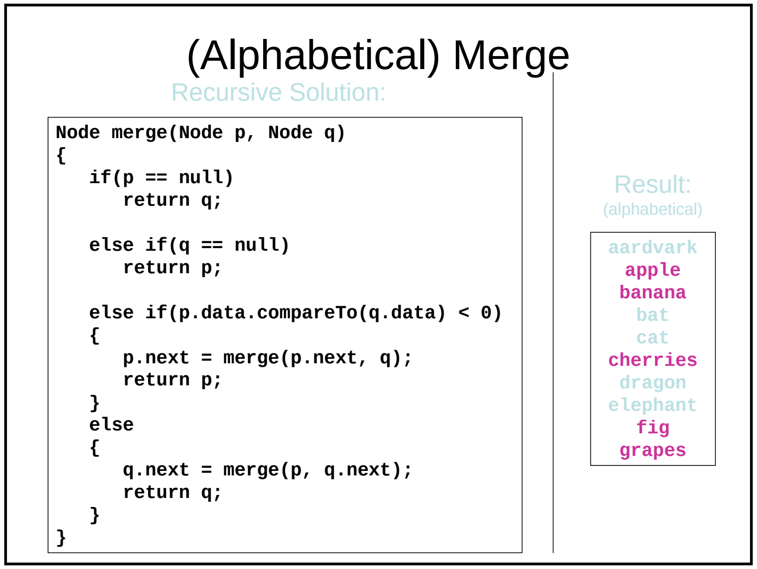
Shuffle

Recursive Solution:

```
Node shuffle(Node p, Node q)
{
   if(p == null)
      return q;

   else if(q == null)
      return p;

   else
   {
      // Note we exchange p and q here
      p.next = shuffle(q, p.next);
      return p;
   }
}
```

Result:

```
apple
aardvark
banana
bat
cherries
cat
fig
dragon
grapes
elephant
```



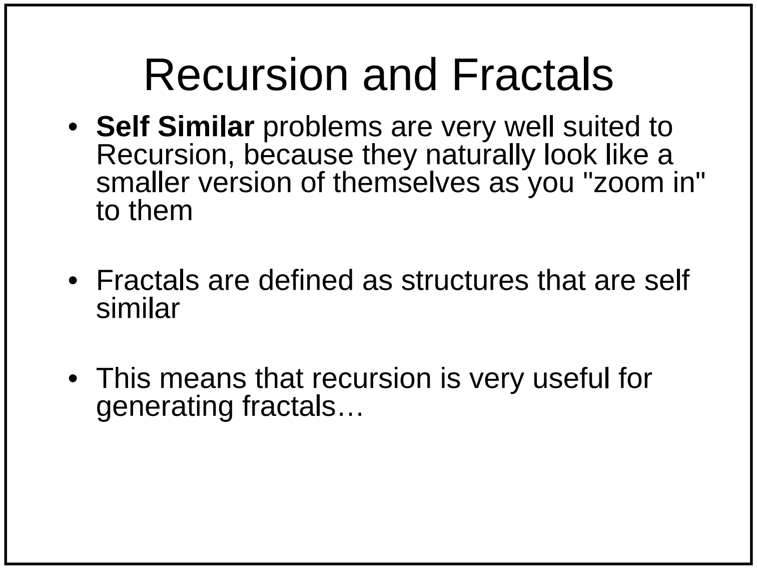
(Alphabetical) Merge

Recursive Solution:

```
Node merge(Node p, Node q)
{
   if(p == null)
      return q;

   else if(q == null)
      return p;

   else if(p.data.compareTo(q.data) < 0)
   {
      p.next = merge(p.next, q);
      return p;
   }
   else
   {
      q.next = merge(p, q.next);
      return q;
   }
}
```

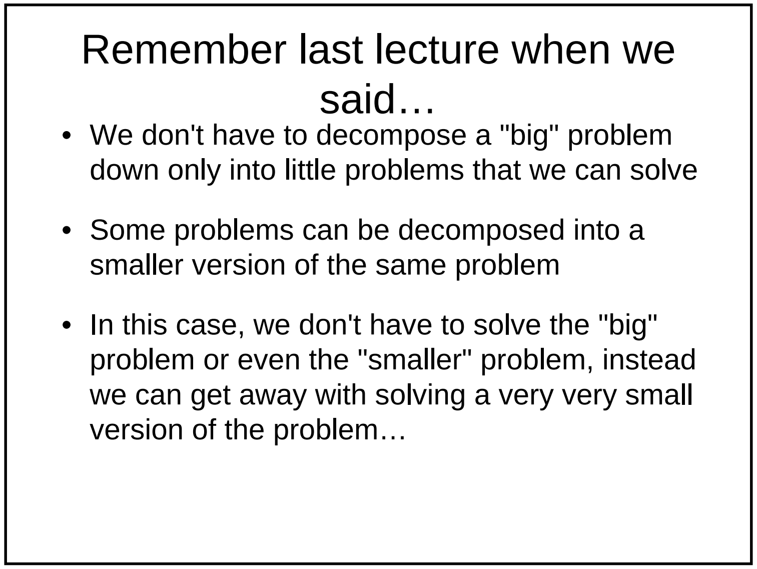
Result:
(alphabetical)

```
aardvark
apple
banana
bat
cat
cherries
dragon
elephant
fig
grapes
```



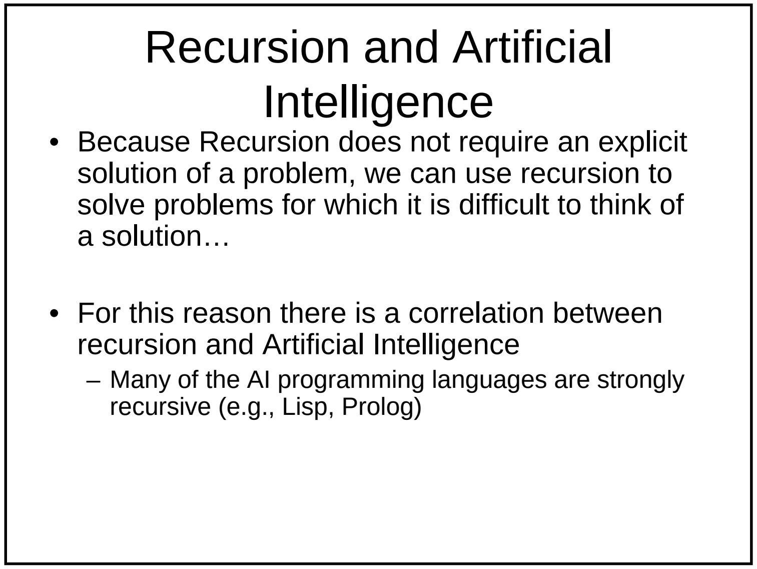
Recursion and Fractals

- **Self Similar** problems are very well suited to Recursion, because they naturally look like a smaller version of themselves as you "zoom in" to them
- Fractals are defined as structures that are self similar
- This means that recursion is very useful for generating fractals…



Remember last lecture when we said…

- We don't have to decompose a "big" problem down only into little problems that we can solve
- Some problems can be decomposed into a smaller version of the same problem
- In this case, we don't have to solve the "big" problem or even the "smaller" problem, instead we can get away with solving a very very small version of the problem…



Recursion and Artificial Intelligence

- Because Recursion does not require an explicit solution of a problem, we can use recursion to solve problems for which it is difficult to think of a solution…
- For this reason there is a correlation between recursion and Artificial Intelligence
  - Many of the AI programming languages are strongly recursive (e.g., Lisp, Prolog)

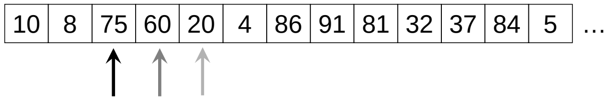
CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

Searching

- Searching is a common problem we often face when writing programs
- The question is, how best to find an item stored in a collection?
- Although the particular data (or Object) we might be looking for could be just about anything, the searching problem itself usually looks about the same

Analysis of Linear Search



10	8	75	60	20	4	86	91	81	32	37	84	5	…

- How long does it take to find our number?
- We could get lucky if the key is near the front
- Otherwise we may have to search all the way to the end of the array
  - This is called the "**Worst Case**", here the worst case = **n** comparisons
- On average we would expect to have to search about half of the array
  - This is called the "**Average Case**", here the average case = ½ **n** comparisons

"Big-O" Notation

- The idea of "Big-O" notation is to provide an idea of the relative time-efficiency of an algorithm
  - We are also worried about memory ("space-efficiency"), but not as much as time-efficiency
- As we just saw, we remove factors that depend only upon the particular implementation (processor, language)
- Terminology:
  - An algorithm like the linear search we just saw, which is $O(n)$, we would say is "Order **n**"

Common Time Complexities

BETTER

$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \times \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time

WORSE

Why are these functions in this order?

Conclusion?

- Although the problem sounds (and is) simple, because the "complexity of our algorithm" is $O(2^n)$ we could never hope to see our program run to completion in our lifetime
- In theoretical terms our goal is to find algorithms and data structures that have a low complexity
- And in terms of applied computer science (i.e., working for "the man") our goal is to know enough about complexity to know which data structure from the API to use (*array* versus *linked-list*) and which sorting algorithm from the API to call to sort our data…

Binary Search

- What if the array of integers was sorted? We could "bisect" the array. Let's find 10….

0						6						12
2	4	5	8	10	14	17	20	22	23	27	29	30
Bottom						Middle						Top

array[Middle] is too large!
So we move "Top" down

Binary Search Analysis

- The binary search algorithm splits the search space in half every iteration
- This means in the worst case it will take $\log(n)$ steps to find the item
- So Binary Search is order: $O(\log n)$

Sorting

- Having sorted data makes searching much faster
- So what options do we have for sorting?
- Let's start with the "Bubble Sort"

Bubble Sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
- Then: Compare each element (except the last two) with its neighbor to the right
  - If they are out of order, swap them
- Then: Compare each element (except the last three) with its neighbor to the right
- Continue as above until you have no unsorted elements on the left

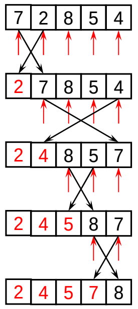
Analysis of Bubble Sort

- The outer loop is executed **n-1** times (call it **n**, that’s close enough)
- Each time the outer loop is executed, the inner loop is executed
- The inner loop executes **n-1** times at first, linearly dropping to just once
- On average, inner loop executes about **n/2** times for each execution of the outer loop
- In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- result is $\mathbf{n} \times \mathbf{n/2} \times \mathbf{k}$, that is, $O(\frac{1}{2}\, n^2 \times k) \approx O(n^2)$

Selection Sort

- Search elements 0 through n-1 and select the smallest
  - Swap it with the element in location 0
- Search elements 1 through n-1 and select the smallest
  - Swap it with the element in location 1
- Search elements 2 through n-1 and select the smallest
  - Swap it with the element in location 2
- Search elements 3 through n-1 and select the smallest
  - Swap it with the element in location 3
- Continue in this fashion until there’s nothing left to search

Selection Sort

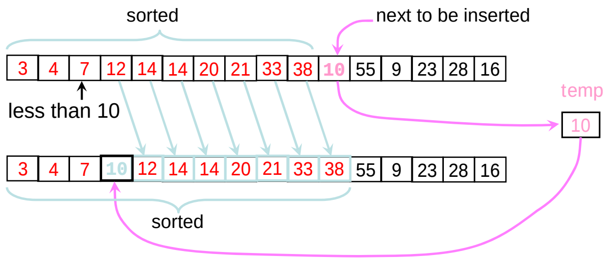


- The outer loop executes **n-1** times
- The inner loop executes about **n/2** times on average (from **n** to **2** times)
- Work done in the inner loop is constant (swap two array elements)
- Time required is roughly **(n-1)×(n/2)**
- This is $O(n^2)$

Insertion Sort

- We have a counter that loops through the array, from bottom to top
- Each new element that the counter points to is inserted in order to the left of the counter
  - This means we have to shuffle elements up the array to make room for each newly sorted element
- Repeat for all elements

One Step of Insertion Sort



Analysis of Insertion Sort

- Runs once through the outer loop, inserting each of **n** elements
- On average, there are **n/2** elements already sorted
- The inner loop looks at (and moves) half of these (this gives a second factor of **n/4**)
- So the time required for an insertion sort to complete sorting the array of **n** elements is proportional to ¼ n^2
- Discarding constants, insertion sort is $O(n^2)$

QuickSort

- Quicksort is one of the fastest sorting algorithms known
- It is naturally a recursive algorithm
- The idea is:
  - Pick any element, and call it "**the pivot**"
  - Re-order the list (in 1 pass) so that all values less than the pivot come before it in the array, and all larger values come after it
  - Recursively sort the two sub-lists (of elements that are smaller than the pivot, and elements that are larger)

Analysis of Quicksort

- The analysis of Quicksort depends upon how lucky the algorithm gets with the pivot values
- If the pivots cause the array to be divided roughly equally every time, then Quicksort is O(nlog n)
- If the pivot values are not lucky, then the Quicksort is order O(n2)
- Although in practice things can be done to ensure that the pivots are chosen well
- And for large sets of data, Quicksort is one of the fastest sorting algorithms we have

Merge Sort

1. Break the set to be sorted in half
2. Use recursion to sort each half
3. Merge the two sorted lists back together

- (For source code see Assignment #8)
- Merge sort works best with:
  - Data where sets can easily be re-ordered (like linked-lists)
  - Analysis:
    - Average Case: $O(n \times \log n)$
    - Worst Case: $O(n \times \log n)$

Sorting Summary

	Average	Worst Case
Bubble	O(n2)	$O(n^2)$
Selection	O(n2)	$O(n^2)$
Insertion	O(n2)	$O(n^2)$
Quicksort	$O(n \times \log n)$	$O(n^2)$
Mergesort	$O(n \times \log n)$	$O(n \times \log n)$

- Quicksort (or variations) are commonly used everywhere, because the worst case is avoidable
- Although it has a poor complexity, insertion sort is fast for very small data sets (small **n**)
- Mergesort is fastest for serially-accessible data

Sorting Summary

- We have covered only the most popular sorting algorithms here
- There are many many more
- But in practice you need to know only four algorithms: ***Insertion*** sort, ***Quicksort***, ***Mergesort***, and the ***HeapSort***
  - Heapsort uses a "Tree" data structure, which you won't cover until next year, and so we can't really discuss it in detail yet (although it's pretty cool, and it's about as fast as Quicksort, although its average case and worst case are both $O(n\times\log n)$).

Next…

The Final Exam