Week 15

Exceptions, Sorting, Searching & Review

Spring 2014

Student Responsibilities

- Reading: Textbook, Chapter 12.1 – 12.3
- Lab is a *participation* lab this week – show up and get a finch to move, light up, and say something, then show me.
- Lab 14 is due **no later than 4:00 pm Friday**
- Attendance
- **Study for finals**: review old exams, quizzes, labs, assigned reading, and handouts — and remember to:

Get some sleep!

Exception Handling – a Short Intro

- There are some runtime situations which cause a Java program to "crash." Examples include:
  - division by zero
  - attempting to access an array with an index which is out of its bounds
  - attempting to dereference a null pointer
- These types of problems are called **exceptions**.
- When a Java method encounters a problem during execution, it responds by **throwing** an exception — this is the process of **reporting an exceptional condition** outside the normal program flow.

Runtime Exceptions

- When an exception is **thrown**, the Java runtime system:
  - **stops executing code** at that point
  - begins **searching backwards** for methods on the control stack — starting with the current method and proceeding to the method which called it, and so forth
  - until it **finds a method that expresses the intent to catch** such an exception if one is thrown
- Most **runtime** exceptions do not need to be caught — the exception will simply propagate backwards on the control stack, eventually causing the program to halt.

`try` & `catch()`

- On the other hand, attempting to access an input file which doesn't exist, for example, is a different situation.
- Programmers who use the `java.io` package are required (i.e., **forced**) by Java to check for the exceptions that the methods in that package throw.
- Thus, the code to open and read a file in Java is not complete unless it explicitly catches exceptions propagated from the `IOException` class.
- The **code that works with data files** must appear inside a **try** statement, with an associated **catch()** statement.

`try - catch()` Syntax

```
try
{
   // code in which an exception might occur
}

catch (type identifier)
{
   // code to respond to the exception
}
```

Where:

- **type** is the type of the exception
- **identifier** is the name of a variable that hold the exception information

File Opening Example

```
try
{
    code to open and read the file
}

catch (IOException ex)
{
    code to respond to exceptions that occur
}
```


InputFile Class by Dr. Mertz — Examples

```
/** Read an entire line from the file. Returns null if an
 * error occurs or the end of the file is reached.
 * @return the next line of text from the file    */

public String readLineFromFile(){
  if (buffer == null)  {       // connection to file failed
    showErrorDialog(NO_FILE_ERROR);
    return null;
  }

  try  {
    return buffer.readLine();
  }

  catch (IOException ex)  {
    showErrorDialog(FILE_IO_ERROR);
    return null;
  }
}
```



```
/** Tries to read next word and convert it into an Integer.
 * The converted value (or null if error) is returned.   */

public Integer readIntegerFromFile()  {
  if (buffer == null) {       // no file connected to buffer
    showErrorDialog(NO_FILE_ERROR);
    return null;
  }

  String word = readWordFromFile();

  if (word == null) { return null; }

  try  { return Integer.valueOf(word);  }

  catch (NumberFormatException ex)  {
    showErrorDialog(word + " is not an integer!");
    return null;
  }
}
```


Searching

- Given a list of items, it is often the case that we need to search the list for a specific value.
- One example might be finding Aunt Sally's address among your list of family and friends.
- Something that impacts how we search is whether or not the list is in order.
- Looking someone up by their last name in a phone book is different than looking for their information on scraps of paper stuffed in your backpack.

Types of Searches

- In a **Linear** search, we start at the beginning and work our way through the entire list one item at a time until we either find what we're looking for, or we reach the end of the list.

  Linear search can be used on any list.

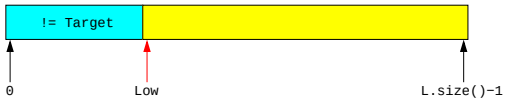
- If a list is sorted, we can use a **Binary** search:
  - check the middle value of the list
  - If we find what we're looking for, we're done.
  - If we don't, then we can throw half the list away and look in the middle of the half–list we now have.
  - This process continues until we find what we're looking for, or the current list is empty.

  Binary search can only be used on sorted lists.

Linear Search Algorithm

```
Mark the region in the list to search

while (the region is not empty) {
   probe the first entry of the region
   if (Target is found){
      note the location
      return true
   }
   else
      go on to the next entry, decrease the region
}

if we get to this point, the region is empty
   yet we still have not found the target, so {
      set location to an impossible value
      return false
   }
```

Linear Search: Marking a Region



Low began at 0. Search is partially completed.

The cyan region to the left of Low has already been considered and does not contain the Target.

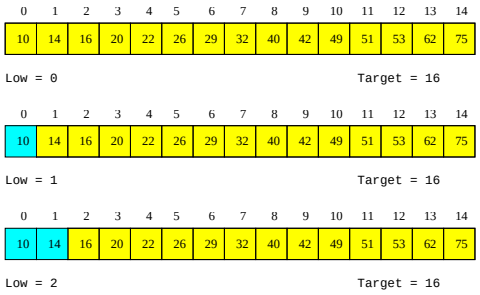
Linear Search: Worst Case Number of Probes

For an ArrayList of size *n*,
linear search makes at most *n* probes (comparisons).

Searching a Temperature List: Linear Search

```
public static boolean LinearSearch(
                  ArrayList<Temperature> L,
                  Temperature Target,
                  Integer Location){
  int Low = 0;
  while (Low < L.size())   {
    // Target is found
    if (Target.isEqual(L.get(Low))
    {
      Location = Low;
      return true;
    }
    else      // Target may be in right-hand section
      Low++;               // Discard mismatch
  }

  Location = L.size();      // Target not in list
  return false;
}
```


Linear Search: Integer List Example



Searching a **Sorted** list: Binary Search Algorithm

```
Mark the region to search

while (the region is not empty) {
  probe near the middle of the region
  if (found)
     note the location, return true
  else
     shrink the region by discarding
     the lower or upper portion as appropriate
}

set location
return false
```


Searching a **Sorted** list: Marking a Region

Low began at 0, High at L.size()-1.
Search is partially completed.

The cyan regions to the left of Low and right of High have already been considered and do not contain the Target.

Binary Search: Worst Case Number of Probes

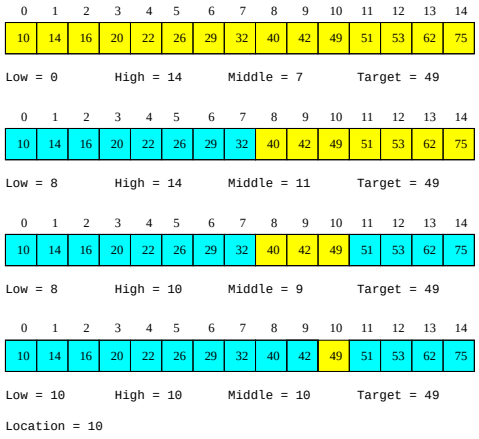
For a list of size n, binary search makes at most $\lfloor \log_2(n) \rfloor + 1$ probes or comparisons.

n	Linear	Binary
$2^2 - 1 = 3$	3	2
$2^3 - 1 = 7$	7	3
$2^4 - 1 = 15$	15	4
$2^5 - 1 = 31$	31	5
$\vdots$	$\vdots$	$\vdots$
$2^{10} - 1 = 1023$	1023	10
$\vdots$	$\vdots$	$\vdots$
$2^{20} - 1 = 1048575$	1048575	20

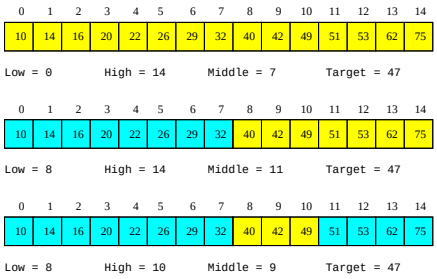
Searching a **Sorted** list: Binary Search

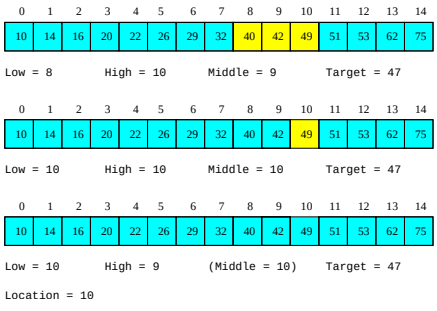
```
int Low = 0;
int High = L.size()-1;
int Middle;

while (Low <= High) {
   Middle = (Low + High)/2;  // Probe near the middle

   if (Target.isEqual(L.get(Middle))){
      Location = Middle;  // Found Target
      return true;
   }
   else if (Target.isLess(L.get(Middle)))
      High = Middle - 1;  // Discard right-hand section

   else  // Target.isGreater(L.get(Middle))
      Low = Middle + 1;   // Discard left-hand section
}

Location = Low;           // Target not in list
return false;
```


Binary Search Example 1

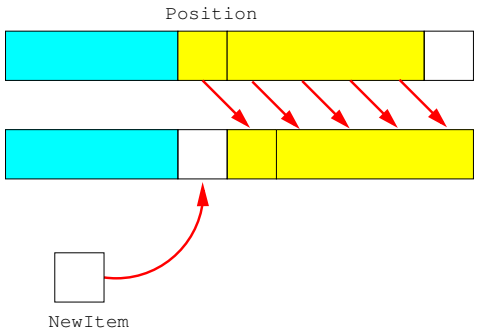


Binary Search Example 2

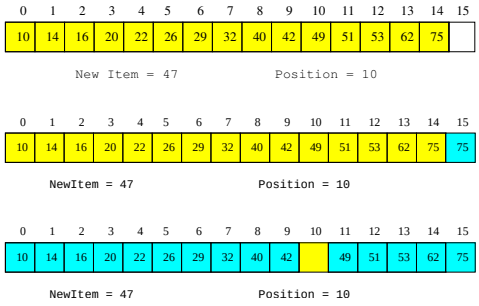




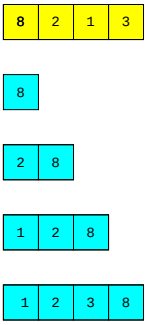
Inserting a New Value into a Primitive Array



Inserting a New Value: Example



Insertion Sort: Example



Using Search and Insert: Insertion Sort with `ArrayList`

```
public static void InsertionSort(ArrayList<Integer> L) {
  // Save a copy of the list
  ArrayList<Integer> Original = new ArrayList<Integer>();

  for (int i = 0; i < L.size(); i++) {
    Original.add(L.get(i));
  }

  // Remove all entries from the parameter vector
  L.clear();

  // Insert back into the given vector from the copy, one
  // element at a time, using Search to find correct location
  int Place;
  for (int i = 0; i < Original.size(); i++) {
    Search(L, Original[i], Place);
    Insert(L, Original[i], Place);
  }
}
```


Course Review

- **Algorithm**
  - Step-by-step method for solving a problem
  - All steps must be unambiguous and executable
  - Must terminate with the correct outcome
- **Object**
  - Encapsulates "state" and "behavior" in one entity
  - examples:
    - a string of characters
    - a rectangle in a graphics system
    - an `Angle` class object

Data Types

- Primitive types
  - `int`
  - `double`
  - `char`
  - `boolean`
  - ...
- Types defined by a Class
  - `String`
  - `GRect`
  - `GPoint`
  - `ArrayList`
  - `Rational`
  - `Integer`
  - ...
- **A reminder**: Don't forget to watch for **integer division**!

Control Structures: Selection

- The `if` statement:

```
if (booleanExpression)
  statement;

if (booleanExpression)
  statement1;
else
  statement2;
```

- `if` statements can be nested
- `switch` statements

Control Structures: Iteration

- the `while` loop

```
while (BooleanExpression)
{
  statements;
}
```

- the `for` loop

```
for (InitExpr; BoolExpr; UpdateExpr)
{
  statements;
}
```

- loops can be nested

Classes

- Composed of **data members** (storage) and **member methods** (messages)
- Members may be **public** or **private** (or protected)
- A class may be **derived** from another class using inheritance
- Examples of classes:
  `GRect`, `String`, `RandomGenerator`, `GSmartCircle`, `Integer`, `ArrayList`, and `Rational`

Messages

Objects of a class can respond to "messages"

- To send a message, we identify a **receiver** (unless inside a class and sending to self)
- ... and the **message**, including any required parameters
- Examples:

```
println(R.getWidth())
R.setColor(Color.BLUE)
rgen.nextInt(low, high)
(getX() < 0 || (getX() + getWidth() > window.getWidth()))
```


Methods

- "Black box view" — **what** do we want the method to do?
- "Glass box view" — **how** does the method achieve its goal?
- The **method header** spells out the interface details:
  - The **accessibility** — public, protected, or private
  - Optional: if the method is `static`
  - The **type of the return value** (`void`, if none)
  - The **name** of the method
  - The details about **parameters**:
    The **order**, **number** and **type** of each parameter
- The implementation spells out the full details of how it does its job
- Invoking methods **inside** (object) vs **outside** (client) a class.

Invoking Methods

- **void** methods typically cause some action to occur
- Methods with a **return value** typically compute some value
- A client of **Static** methods in a class must use the class name as the "receiver"
- Some examples:

```
char c = Character.toUpperCase(SomeChar);
double y = Math.sqrt(x);

int z = myMin(a, b, c);

Display(a, b, c);
Collections.sort(MyList);
```


ArrayList

- A contiguous list of items, all of the same type
- First valid index of `ArrayList X` is `0`
- Last valid index is `X.size()-1`
- Common messages:
  - `size()`
  - `clear()`, `remove(ndx)`
  - `add(value)`
  - `get(ndx)`
  - `set(ndx, value)`

Searching

- Linear search sequentially probes a search region, comparing its items to the target
- Binary search reduces the search region by half on each probe
- unsorted `ArrayList` — linear search
- sorted `ArrayList` — binary search

Sorting

- There are **many** different sorts —
  - Insertion
  - Selection
  - Bubble
  - Heap
  - Quick
  - Bucket, etc. (**Advertisement**: take MAT 2670!)
- Insertion sort uses binary search to locate the insertion position.

Design A Header For A Method Which. . .

- Tests whether or not a given integer is a prime number.
- Determines the number of prime numbers k in the closed interval $[L, U]$.
- Determines the number of radians in an angle which is specified in degrees.
- Replaces every occurrence of a blank in a `String` with a hyphen.

- Replaces every occurrence of one specified character in a `String` with another character.
- Determines the number of hyphens in a given `String`.
- Determines the midpoint of two specified points in the plane.

Design An Implementation For A Method Which. . .

- Given a sorted `ArrayList A`, creates an `ArrayList B` which is sorted in reverse order
- Given two points, `P1` and `P2`, finds the point which is `r`% of the way from `P1` to `P2`.

- Given a `String S`, modifies it so all alphabetic characters in it are lowercase
- Given `C`, an `ArrayList` of characters, modifies it so all alphabetic characters in it are lowercase

Designing Derived Classes

- Desired: A class derived from the GLine class to create a rotating line
- Data Members: a pause time, dx, and dy (for the end point of the line)
- Constructor : start x, start y, end x, end y, pause time, dx, dy, and color
- Constructor : end x, end y, and defaults of start x and start y equal 0.0, pause time of 50, dx and dy of 2.0, and color blue
- move() : reset the end point by dx and dy, then pause `setEndPoint()` and `getEndPoint()` are available

Designing A Class From Scratch

- Desired: A new class which represents the menu items at a fast food place, and a list of such items
- Item: name, carbs, calories, cost with inspectors and mutators
- List: of Items with methods: total items, total carbs, total calories, total cost