**Lecture 14 - Chapter 6: Arrays – Mon Oct 9 or Tues Oct 10**

**Announcements**

Reading:

* Chapter 6

Assignments:

* Assign: Assignment #6 - due on **Oct 18** (MW class) or **Oct 18** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. Searching Arrays
2. Variable Length Arrays
3. Secure C Programming

**Today’s Terminology**

**Terminology**

* Array
  + A data structure used to store a collection of values that are all the same type
* Index
  + Refers to a specific element within an array
  + Must be an integer or integer expression
  + The **first element** in an array is at index 0, the second element is at index 1, etc.
* Indexed Variable
  + Used to reference each element in an array
* Array Initializer
  + A statement where you define and initialize an array
* Out of Bounds Error
  + Attempting to access an element with an index outside the range of the array is a runtime error
* Off by One Error
  + Mistakenly referencing the first element in an array with the index 1, not index 0
* Linear Search
  + Searching an array for a specific key value by starting at beginning and comparing the key with each element sequentially.
* Binary Search
  + Searching a pre-sorted array by splitting the array in half on each comparison.
* Bubble Sort
  + Combination of searching and sorting. During each pass, the largest element is moved to its proper location in the array.
* Big O Notation
  + A way to describe the performance of an algorithm

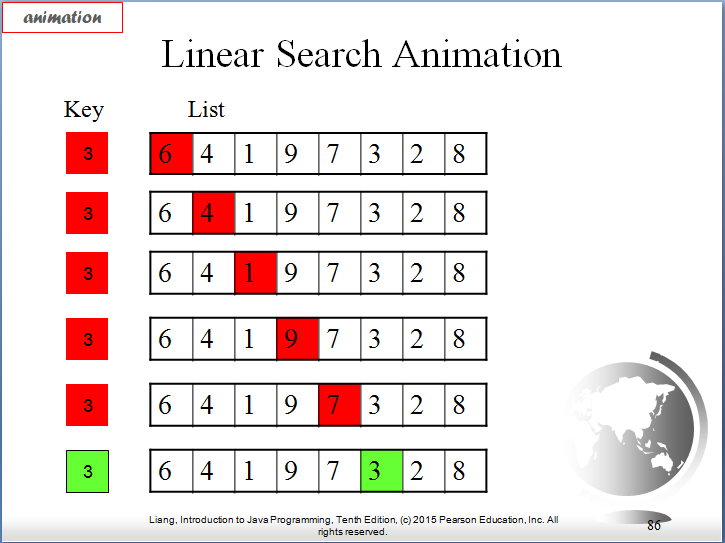
**Searching Arrays**

**Searching**

* In computer programming another common activity is to write code to find a specific value in an array
* Two techniques we will look at are
  + Linear search
  + Binary search

**Linear Search**

* Simple search – not very efficient algorithm!
* Starting at beginning of an array and look for a specific value one element at a time
* When a match is found that index of the match is returned, otherwise a -1 is returned
* The elements in the array are in any order
* Good for small or sorted arrays
* Execution time grows linearly as the array size grows!
  + Best case: 1st element is key we are looking for
  + Worst case: last element is key or doesn't find key
  + Average: 50% of elements are examined before key is found
* Visually



* Code for linear search:

// Finds a key value in an array

// Input: integer array, key value to search for, length of array

// Output: index or key if found, otherwise -1

size\_t **linearSearch** (**const** **int** array[], **int** key, size\_t size) {

**for** (size\_t i = 0; i < size; i++) {

**if** (array[i] == key) {

**return** i;

}

}

**return** -1;

} //linearSearch

* Trace
  + Assume we have the following in code in main

**int** **main**(void) {

**int** arrayToSearch[100];

// Fill array with 100 random numbers between 0 and 99

**for** (size\_t i = 0; i < 100; i++) {

arrayToSearch[i] = **rand**() % 100;

}

// Ask user for the key to look for in the array

**puts** ("What value would you like to search for?");

**int** key;

**scanf**("%d", &key);

// Perform a linear search for the key

size\_t searchResult = linearSearch(arrayToSearch, key, 100);

// Check result from search

**if** (searchResult != -1) {

**printf** ("%d was found in location %d\n", key, searchResult);

}

**else** {

**printf** ("%d was not found in the array\n", key);

}

// Prove search result worked by printing the array

**puts** ("Showing proof the search worked correctly");

printArray(arrayToSearch, 100);

} // main

* + The array is created
  + The array is filled with random values between 0 and 99
  + The user is prompted for the "key" value to search array for
  + The array and key are sent to the **linearSearch**
  + The **linearSearch** examines array elements sequentially looking for "key"
  + The **linearSearch** returns either the index or the key or a -1
  + The output is shown below:

**Displays**

What value should we search for?

81

81 was found in location 19

Printing all elements to show proof our search worked correctly

arrayToSearch[0] = 94

arrayToSearch[1] = 58

arrayToSearch[2] = 54

arrayToSearch[3] = 94

arrayToSearch[4] = 75

arrayToSearch[5] = 40

arrayToSearch[6] = 72

arrayToSearch[7] = 48

arrayToSearch[8] = 56

arrayToSearch[9] = 7

arrayToSearch[10] = 64

arrayToSearch[11] = 71

arrayToSearch[12] = 70

arrayToSearch[13] = 75

arrayToSearch[14] = 61

arrayToSearch[15] = 51

arrayToSearch[16] = 29

arrayToSearch[17] = 44

arrayToSearch[18] = 96

arrayToSearch[19] = 81

arrayToSearch[20] = 82

arrayToSearch[21] = 34

arrayToSearch[22] = 45

.

.

.

arrayToSearch[94] = 40

arrayToSearch[95] = 32

arrayToSearch[96] = 87

arrayToSearch[97] = 10

arrayToSearch[98] = 66

arrayToSearch[99] = 76

What value should we search for?

7

7 was not found in the array

Printing all elements to show proof our search worked correctly

arrayToSearch[0] = 17

arrayToSearch[1] = 28

arrayToSearch[2] = 77

arrayToSearch[3] = 14

arrayToSearch[4] = 79

arrayToSearch[5] = 93

arrayToSearch[6] = 10

arrayToSearch[7] = 5

**.**

**.**

**.**

arrayToSearch[92] = 24

arrayToSearch[93] = 27

arrayToSearch[94] = 22

arrayToSearch[95] = 56

arrayToSearch[96] = 73

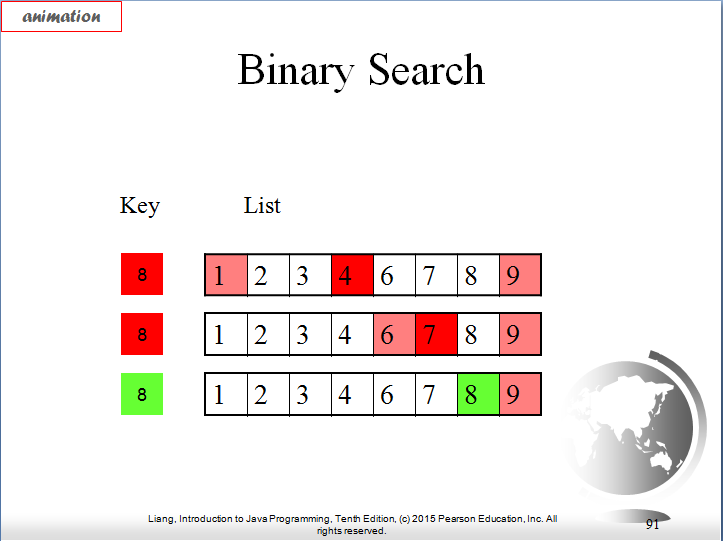
arrayToSearch[97] = 84

arrayToSearch[98] = 30

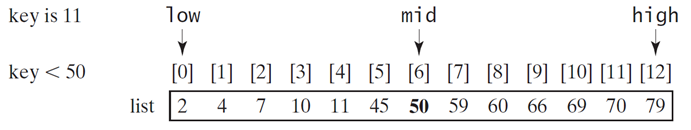
arrayToSearch[99] = 96

**Binary Search**

* More efficient but more complex than linear search
* Another approach but it requires that the ***elements in the array to be ordered*** before the search begins
* Starts with element in the ***middle*** of the array
  + If key is less than "middle" element, then only search for key in ***1st half of array***
  + If key is equal to "middle" element, return match!
  + If key is greater than "middle" element, then only search for key in ***2nd half of array***
* When a match is found that index of the match is returned, otherwise a -1 is returned
* Efficient algorithm - array shrinks by 50% after each comparison!
  + Assume the array contains 1024 elements (210)
    - After 1st comparison - reduced search to 1/2 of the elements (512)
    - After 2nd comparison - reduced search to 1/2 of that 1st half (256)
    - After 3rd comparison - 128
    - After 4th comparison - 64
    - After 5th comparison - 32
    - After 6th comparison - 16
    - After 7th comparison - 8
    - After 8th comparison - 4
    - After 9th comparison - 2
    - After 10th comparison - 1
    - On 11th comparison 1 element left to compare with key
  + Compared to linear search
    - Best case: 1 search
    - Worse case: 1024 comparisons
    - Average: 50% of elements are examined before key is found - 512 comparisons!
* Visually



* Code for binary search
  + Do this incrementally because it's a bit confusing
  + **Let's do it for 1 comparison (NOT COMPLETE VERSION!)**
  + Assume
    - Array has 13 values
    - Array is ordered in ascending order
    - Key is 11



size\_t **binarySearch** (**const** **int** array[], **int** key, size\_t low, size\_t high) {

// Find the index for middle element

**int** middle = (low + high)/2;

**if** (key == array[middle]) {

**return** middle;

}

**else if** (key < array[middle]) {

high = middle - 1;

}

**else** **if** (key == array[middle]) {

**return** middle;

}

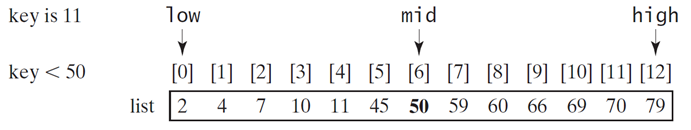
**else** **if** (key >= array[middle]) {

low = middle + 1;

}

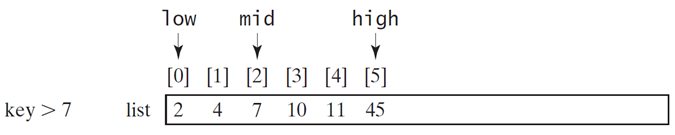
} // binarySearch

* Trace 1st incremental step of the code
  + **Remember this code is NOT the solution yet - it's a step towards the solution!!!**
  + Starting with index for
    - low = 0
    - high = 12
    - middle = 6



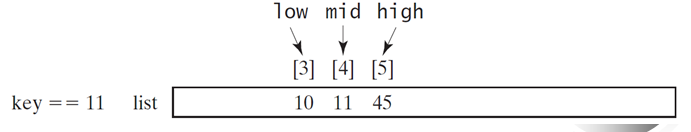
Since key < 50, ***low*** remains at 0, ***high*** = middle - 1 = 5

* + On **2nd comparison** we now have only the 1st half of the array left to search
    - low = 0
    - high = 5
    - middle = 2 (5/2 = 2.5 but integer division so becomes 2)



Since key > 7, ***low*** = middle + 1 = 3, ***high*** remains at 5

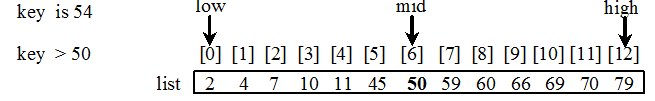
* + On **3rd** **comparison** we have
    - low = 3
    - high = 5
    - middle = 4 (8/2 = 4)



Since key = 11 we hit the statement

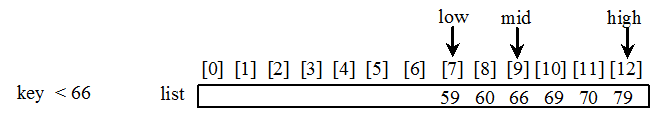
**return** middle;

* + Need to add a loop to do test over and over until there is only 1 element to compare!
  + Question:
    - How do we know when we can stop comparing?
    - Assume in above example we weren't looking for key = 11, say looking for key = 54
      * low = 0
      * high = 12
      * middle = 6



Since key > 54, ***low*** = middle + 1 = 7, ***high*** remains at 12

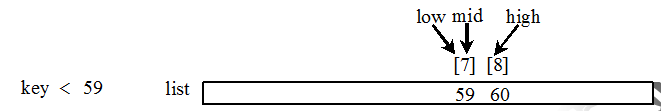
* + - On **2nd comparison** we now have only the 2nd half of the array left to search
      * low = 7
      * high = 12
      * middle = 9 (17/2 = 9.5 but integer division so becomes 9)



Since key < 66, ***low*** remains at 7, ***high*** = middle - 1 = 8

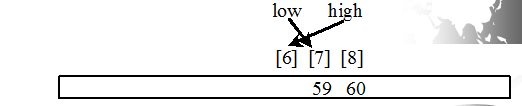
* + - On **3rd** **comparison** we have
      * low = 7
      * high = 8
      * middle = 7 (15/2 = 7.5)

***middle*** = 8 + 7 = 9.5 = 7.5 = 7



Since key < 59, ***low*** remains at 7, ***high*** = middle - 1 = 6

* + - On **3rd** **comparison** we have
      * low = 7
      * high = 6 Key to knowing value is not in array!!!



* Key to knowing we can stop search
  + When low is greater than high!!

* Adding loop to code

// Finds a key value in an array using a binary search

// Input: integer array, key value to search for, low index, high index

// Output: index of key if found, otherwise -1

size\_t **binarySearch** (**const** **int** array[], **int** key, size\_t low, size\_t high) {

// Continue searching as long as index of highest element is

// bigger than or equal to index of lowest element in array

**while** (low <= high) {

// Find middle element of the array

size\_t middle = (low + high)/2;

**if** (key == array[middle]) {

**return** middle;

}

**else** **if** (key < array[middle]) {

high = middle - 1;

}

**else** {

low = middle + 1;

}

} // while

**return** -1; // Key not found

} //binarySearch

* Trace
  + Assume we have the following code in main

**int** **main**(void) {

// Perform a binary search

**puts** ("Binary Search");

**int** arrayToBinarySearch[12] = {2, 4, 6, 7, 11, 34, 56, 78, 102, 134, 155, 345};

// Example where key is in the array

**int** key = 11;

**int** binarySearchResult = binarySearch(arrayToBinarySearch, ke1, 0, 12);

**if** (binarySearchResult != -1) {

**printf** ("%d was found in location %d\n", key, binarySearchResult);

}

**else** {

**printf** ("%d was not found in the array\n", key);

}

// Example where key is not in the array

key = 54;

binarySearchResult = binarySearch(arrayToBinarySearch, key, 0, 12);

**if** (binarySearchResult != -1) {

**printf** ("%d was found in location %d\n", key, binarySearchResult);

}

**else** {

**printf** ("%d was not found in the array\n", key);

}

} // main

**Displays**11 was found in location 4

54 was not found in the array

**Notes**

* The array must already be ordered to use binary search!
* Binary search cuts the search in 1/2 for each comparison
* Works if there are duplicates in the list
* You can return
  + -1 to indicate value is not in array or
  + location of where the value would be inserted into the array
* Linear search is good for:
  + Small arrays
  + Unsorted arrays
* Binary search is good for:
  + Large arrays that are presorted

**Variable Length Arrays**

**Variable Length Arrays**

* C Standard (C99) allows you to handle arrays of unknown size using variable-length arrays (VLAs)
* Size is defined in terms of an expression evaluated at execution time

**int** arraySize;

**puts** ("Enter the size of the array");

**scanf** ("%d", &arraySize);

// Define an array with the size the user will enter

**int** array[arraySize];

// Put values into the array

**for** (size\_t i = 0; i < arraySize; i++) {

array[i] = i;

}

// Print out values in the array

**for** (size\_t i = 0; i < arraySize; i++) {

**printf** ("array[%d] = %d\n", i, array[i]);

}

**Displays**

array[0] = 0

array[1] = 1

array[2] = 2

array[3] = 3

array[4] = 4

array[5] = 5

array[6] = 6

array[7] = 7

array[8] = 8

array[9] = 9

array[10] = 10

array[11] = 11

array[12] = 12

array[13] = 13

array[14] = 14

* NOTE: not supported in Microsoft Visual C++

**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other lanaguages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**Bounds Checking for Array Indices**

* C does not provide bounds checking
* You must check that your access to an array is within the array and not out of bounds
* Reading or writing outside the bounds is a common security flaw
* Accessing elements out of bounds can lead to:
  + Programs crashing
  + Program executing using bad data
  + Corrupting data

**scanf\_s**

* Strings are arrays so bounds checking is important with string processing
* Again, C does not provide bounds checking when reading a string into a string array
* If incoming string is too big:
  + scanf will write beyond end of the array
* C11 standard, optional Annex K
  + Provides new, more secure versions of string processing functions
  + Not all compilers support, if yours does, then should use updated functions
  + scanf\_s
    - ensures incoming string does not write beyond array
    - scanf

**Don’t Use Strings Read from User as Format-Control Strings**

* printf (format-string, list-of-expressions);
* printf (“The sting entered by the user is %s”, someStringEnteredByUser);