Revised: 5/7/2025

Arrays II

Michael C. Hackett

Assistant Professor, Computer Science

Lecture Topics

- Resizing an Array
- Testing Array Equality
- Multidimensional Arrays
- Search Algorithms
  - Linear Search
  - Binary Search

Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **`Consolas`**
Output – `Courier New`

Boolean expression is false

Boolean expression is true

Resizing an Array

- To expand the length of an array:
  1. Create a second, temporary array with a longer length than the original.
  2. Deep copy the contents of the shorter array into the temporary array.
  3. Shallow copy the temporary array to the original's variable.
     - This will replace the original array, with the new bigger array.
  4. Set the temporary variable to null.
     - The variable no longer needs to reference the array.

Resizing an Array

```
    int[] original = {3, 5, 7, 9};
1   int[] temporary = new int[original.length + 2];

    for(int i = 0; i < original.length; i++) {
2      temporary[i] = original[i];
    }

3   original = temporary;
4   temporary = null;
```

Before	After
3, 5, 7, 9	3, 5, 7, 9, 0, 0

When making an array larger, new indexes are given the following default values:

- 0 (number type arrays)
- '' (char type arrays)
- false (boolean type arrays)
- null (object arrays)

Resizing an Array

- To shrink the length of an array:
  1. Create a second, temporary array with a shorter length than the original.

  2. Deep copy the contents of the longer array into the temporary array.
     - Not all will fit.

  3. Shallow copy the temporary array to the original's variable.
     - This will replace the original array, with the new smaller array.

  4. Set the temporary variable to null.
     - The variable no longer needs to reference the array.

Resizing an Array

```
    int[] original = {3, 5, 7, 9};
1   int[] temporary = new int[original.length - 2];

    for(int i = 0; i < temporary.length; i++) {
2       temporary[i] = original[i];
    }

3   original = temporary;
4   temporary = null;
```

Before	After
3, 5, 7, 9	3, 5

Testing Equality of Arrays

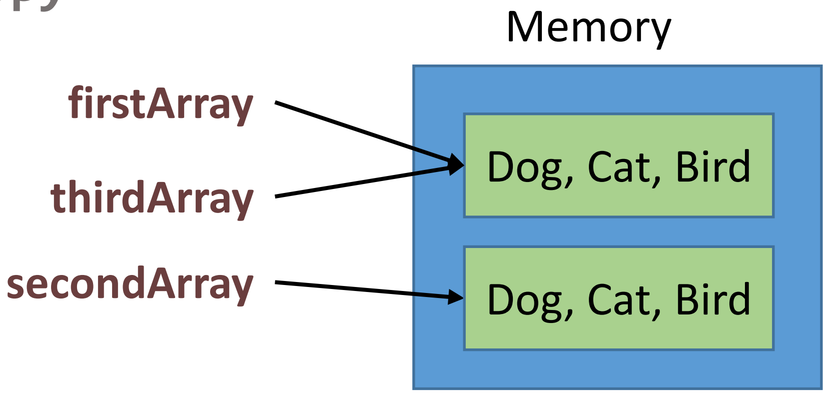
- Using the equality operator (==) to compare arrays only tests if the *reference* is equal, <u>not</u> the values/data.
  - In other words, == only tests if the two array variables are shallow copies.

```
String[] firstArray = {"Dog", "Cat", "Bird"};
String[] secondArray = {"Dog", "Cat", "Bird"};
String[] thirdArray = firstArray; //Shallow Copy

                true
if(firstArray == thirdArray) {

}
                false
if(firstArray == secondArray) {

}
```



Testing Equality of Arrays

- Comparing equality of two arrays is normally done with a one-to-one comparison.
  - Index 0 of both arrays match, index 1 of both arrays match, and so on.

```
int[] firstArray = {3, 5, 7, 9};
int[] secondArray = {3, 5, 7, 9};

boolean equal = true;

for(int i = 0; i < firstArray.length; i++) {
    if(firstArray[i] != secondArray[i]) {
        equal = false;
        break;
    }
}
```

Testing Equality of Arrays

- Two arrays are typically not equal if they don't have the same number of elements.
  - Checking they have equal lengths will also prevent an ArrayIndexOutOfBoundsException.

```
int[] firstArray = {3, 5, 7, 9};
int[] secondArray = {3, 5, 7};
boolean equal = true;
if(firstArray.length == secondArray.length) {
    for(int i = 0; i < firstArray.length; i++) {
        if(firstArray[i] != secondArray[i]) {
            equal = false;
            break;
        }
    }
}
else {
    equal = false;
}
```

Multidimensional Arrays

- When an array contains arrays, it is called ***multidimensional***.

  - A one dimensional array:

```
int[] my1DArray = {2, 4, 6};
```

  - A two dimensional array:

```
int[][] my2DArray = {{8, 3, 7}, {1, 9, 9}, {5, 6, 9}};
```

Multidimensional Arrays

- It’s often better to write two dimensional arrays like this:

```
int[][] my2DArray = {{8, 3, 7},
                     {1, 9, 9},
                     {5, 6, 9}};
```

- This way, it’s easier to see each “row” (first dimension) and “column” (second dimension).

Multidimensional Arrays

- Empty two dimensional arrays are initialized by specifying the number of rows (first) and columns (second):

```
int[][] my2DArray = new int[3][4];
```

Multidimensional Arrays

- Elements in a two dimensional array are referenced by row and column:
  - Row and column numbers start at zero.

```
int[][] my2DArray = {{8, 3, 7},
                     {1, 9, 9},
                     {5, 6, 9}};

my2DArray[1][2] = 2; //Assignment
System.out.println(my2DArray[0][1]); //Retrieval/Prints 3
```

Multidimensional Arrays

```
int[][] my2DArray = {{2, 4, 6},
                     {1, 3, 5},
                     {3, 6, 9},
                     {1, 2, 3}};
```

What element is at `my2DArray[0][2]`?

What element is at `my2DArray[3][1]`?

What element is at `my2DArray[1][0]`?

Multidimensional Arrays

- Rows in a multidimensional array do not have to be the same length.
  - This is called a ***Ragged Array***.

```
int[][] my2DArray = {{2, 4, 6},
                     {1, 3},
                     {9},
                     {1, 2, 3, 4}};
```

- Be careful with ragged arrays as not all rows have the same number of columns.

`my2DArray[2][1]` does not exist, even though every other row has a column 1.

Multidimensional Arrays

- Two for loops are required to iterate through a two dimensional array.

```
int[][] my2DArray = {{8, 3},
                     {1, 9}};

for(int i = 0; i < my2DArray.length; i++) {
   for(int j = 0; j < my2DArray[i].length; j++) {
      System.out.println(my2DArray[i][j]);
   }
}
```

Rows

Columns

Multidimensional Arrays

- Iteration through a two dimensional array using enhanced for loops.

```
int[][] my2DArray = {{8, 3},
                     {1, 9}};
```

Rows

```
for(int[] row : my2DArray) {
   for(int col : row) {
      System.out.println(col);
   }
}
```

Columns

Multidimensional Arrays

- There is no limit to the number of dimensions an array can have.

- A three dimensional array:

```
int[][][] my3DArray = {{{4,8},{15,16,23,42}},{{11,33},{22,44}}};
```

- In the case of a three dimensional array, the rows themselves have rows.

```
int[][][] my3DArray = new int[2][2][3];
```

Rows

Rows in each row

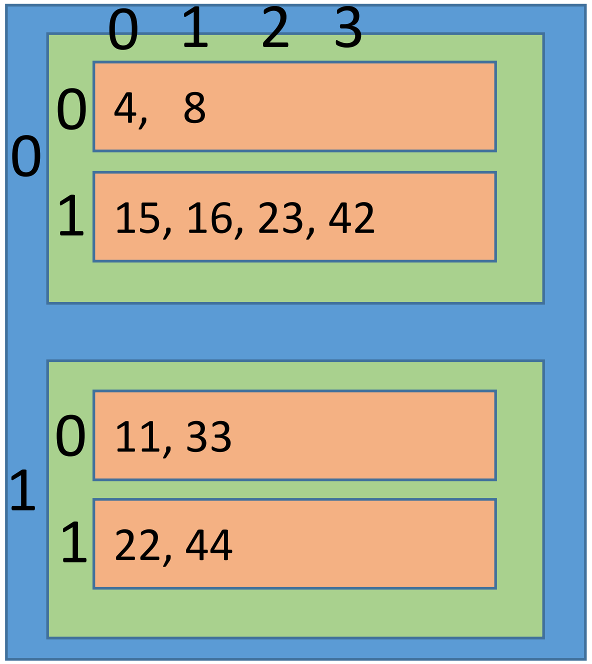
Columns

Multidimensional Arrays

```
                                  0    1     2    3
int[][][] my3DArray = {{{4, 8},0
                          {15,16,23,42}},1          0
                         {{11,33}, 0
                          {22,44}}}; 1              1
```

What element is at `my3DArray[0][1][2]`?

What element is at `my3DArray[1][0][0]`?



Multidimensional Arrays

- Three for loops are required to iterate through a three dimensional array.

```
int[][][] my3DArray = {{{4, 8},
                         {15,16,23,42}},
                        {{11,33},
                         {22,44}}};

for(int i = 0; i < my3DArray.length; i++) {
   for(int j = 0; j < my3DArray[i].length; j++) {
      for(int k = 0; k < my3DArray[i][j].length; k++) {
         System.out.println(my3DArray[i][j][k]);
      }
   }
}
```

Columns

Inner Rows

Outer Rows

Multidimensional Arrays

- Iteration through a three dimensional array using enhanced for loops.

```
int[][][] my3DArray = {{{4, 8},
                        {15,16,23,42}},
                       {{11,33},
                        {22,44}}};
```

```
for(int[][] outerRow : my3DArray) {
    for(int[] innerRow : outerRow) {
        for(int column : innerRow) {
            System.out.println(column);
        }
    }
}
```

Columns

Inner Rows

Outer Rows

Linear Search (Sequential Search)

- A ***search algorithm*** is a series of steps that, when followed, tries to locate and/or retrieve information a set of data (ie. arrays and lists).

- A linear search begins searching at the beginning of an array (index 0) and continuing until the item is found.

- Check index 0; if the element is not what you are looking for, continue to index 1; if the element is not what you are looking for, continue to index 2 (and so on…)

Linear Search (Java code)

- Checking to see if an array of ints contains the number 50.

```
int foundIndex = -1;

for(int i = 0; i < array.length; i++) {
   if(array[i] == 50) {
      foundIndex = i;
      break;
   }
}
```

Since we found what we needed,
we can exit the loop.

Linear Search

- Order of the elements (alphabetical, numerical, etc.) does not effect searching.

- Best case scenario: The information sought is the first element.

- Worst case scenario: The information sought is the last element.

Binary Search

- Takes a “divide and conquer” approach.

- Begins searching in the middle of the array or list.
  - If the middle element is not what we are looking for, we then split the array/list in half:
    - If the value sought is greater* than the middle element, we will then check the middle element of the second half of the array/list.
    - If the value sought is less* than the middle element, we will then check the middle element of the first half of the array/list.
  - The process begins again with the new half.

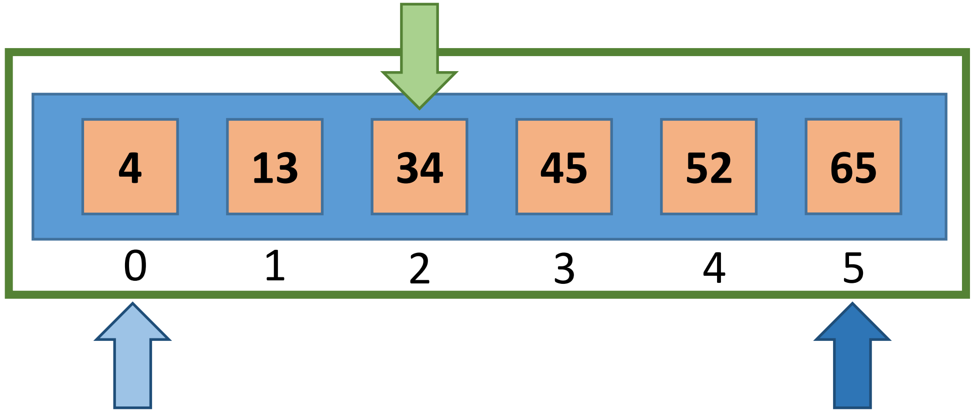
*- The array must be in some order/sequence! (Alphabetically, numerically, etc.)

Binary Search

- We must keep track of a few index values:
  - The middle index – the index in the middle value of the lower and upper boundaries.
  - The lower boundary – the lowest index of the portion of the list we are searching.
  - The upper boundary – the highest index of the portion of the list.
- After each iteration of the algorithm, we recalculate:
  - The lower **or** upper boundary.
  - The middle index.

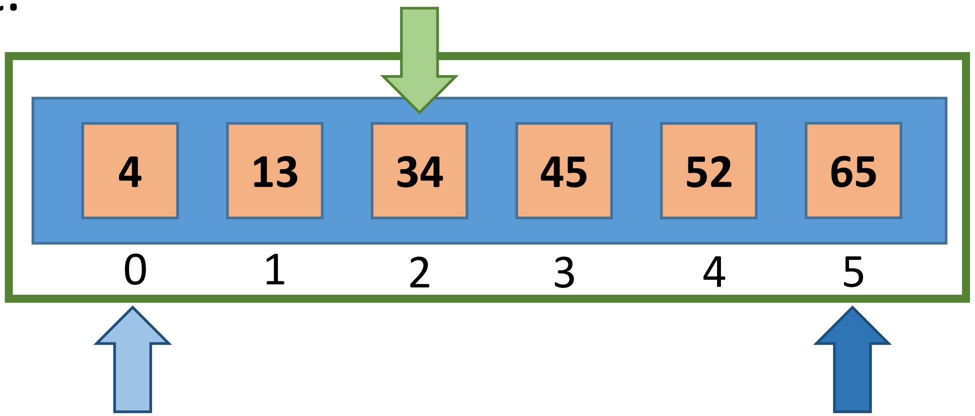
Binary Search (Searching for 45)

- In the first step of the algorithm:
  - The lower boundary is index 0.
  - The upper boundary is the last index.
  - To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: (0 + 5)/2 = 2.5 -> 2



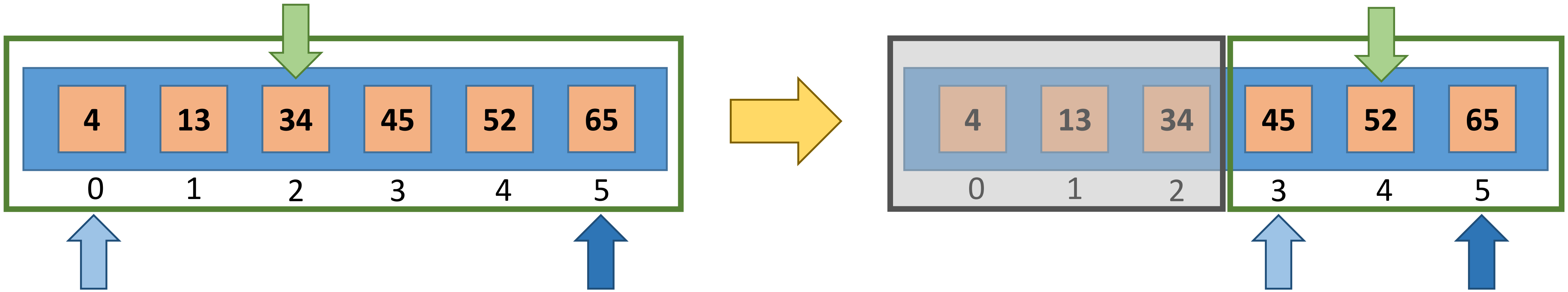
Binary Search (Searching for 45)

- Next, we do one of three things:
  - If the value we are seeking (45) is at this middle index, we are done searching.
  - **If the value we are seeking is greater than this value, then we will search the upper half of this list.**
  - If the value we are seeking is less than this value, then we will search the lower half of this list.



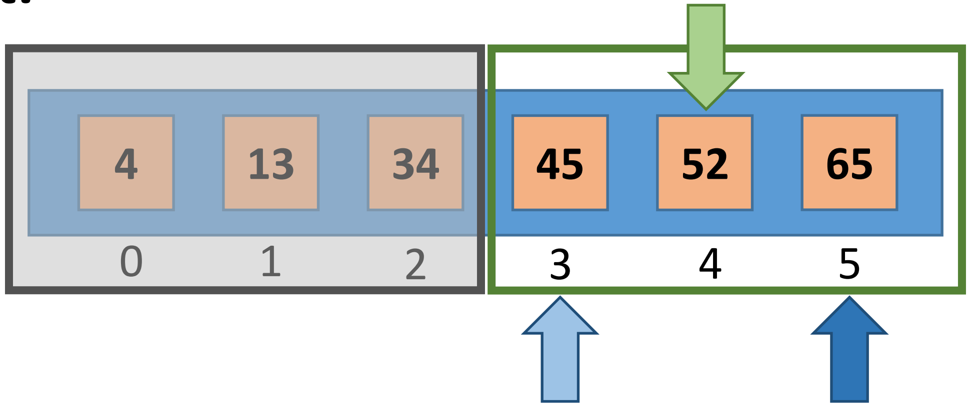
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Middle Index + 1 -> 2 + 1 = 3
  - Upper Boundary: Does not change.
  - Middle Index: (Lower + Upper) / 2 -> (3 + 5) /2 = 4



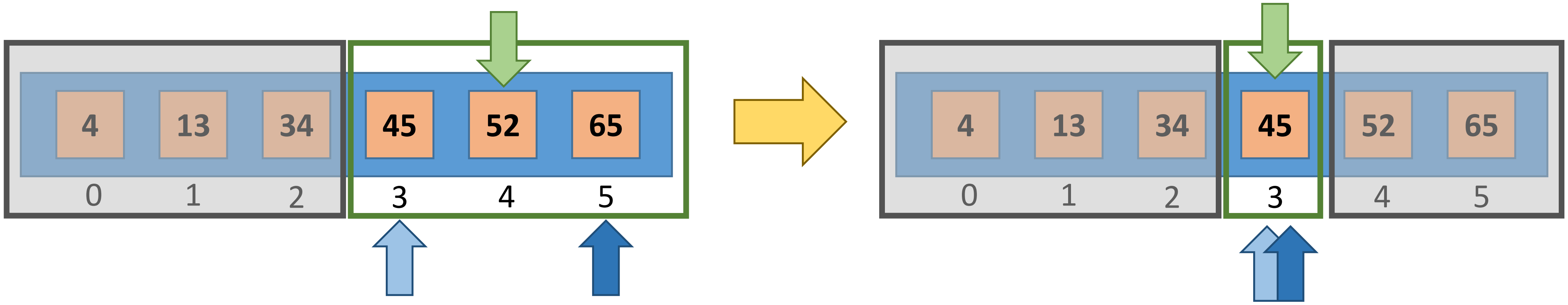
Binary Search (Searching for 45)

- We start the process over:
  - If the value we are seeking (45) is at this middle index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this list.
  - **If the value we are seeking is less than this value, then we will search the lower half of this list.**



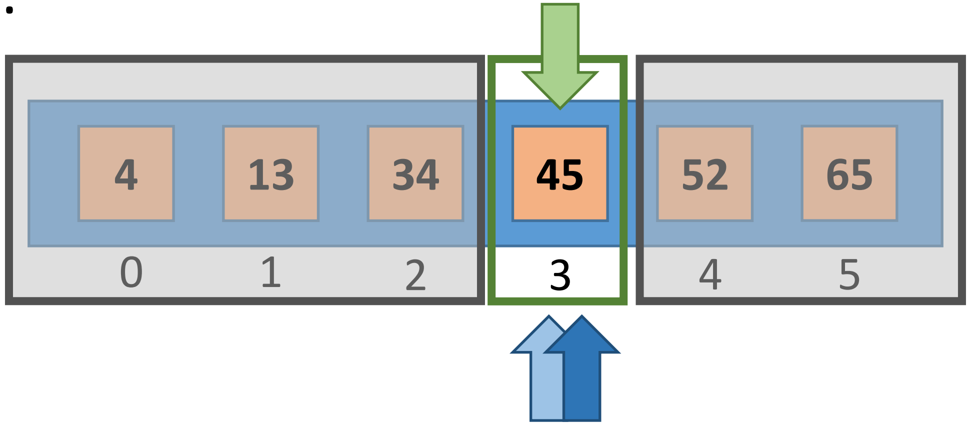
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Does not change.
  - Upper Boundary: Middle Index - 1 -> 4 - 1 = 3
  - Middle Index: (Lower + Upper) / 2 -> (3 + 3) /2 = 3



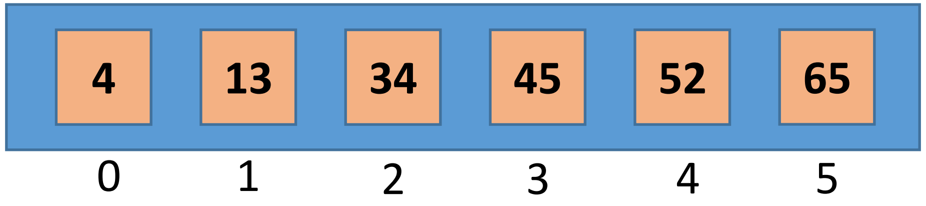
Binary Search (Searching for 45)

- We start the process over:
  - **If the value we are seeking (45) is at this index, we are done searching.**
  - If the value we are seeking is greater than this value, then we will search the upper half of this list.
  - If the value we are seeking is less than this value, then we will search the lower half of this list.



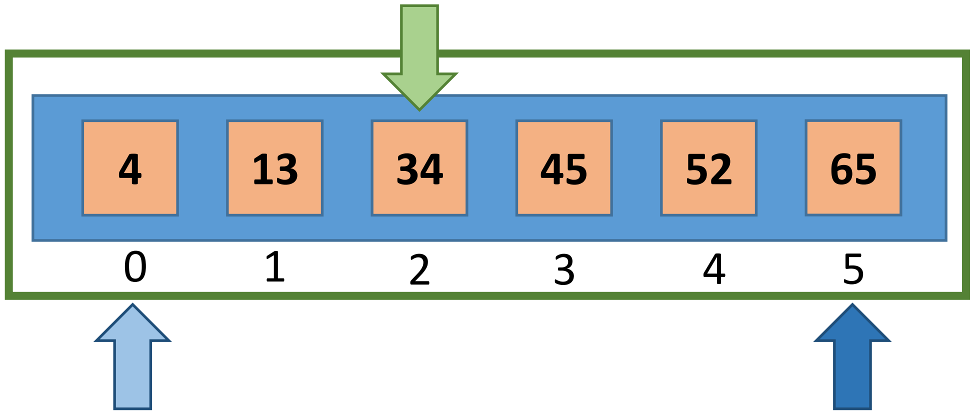
Binary Search

- What happens when the value we are looking for isn't in the array/list?
  - How does the algorithm know when to stop halving the array/list?
  - **When the upper boundary index is less than the lower boundary index.**



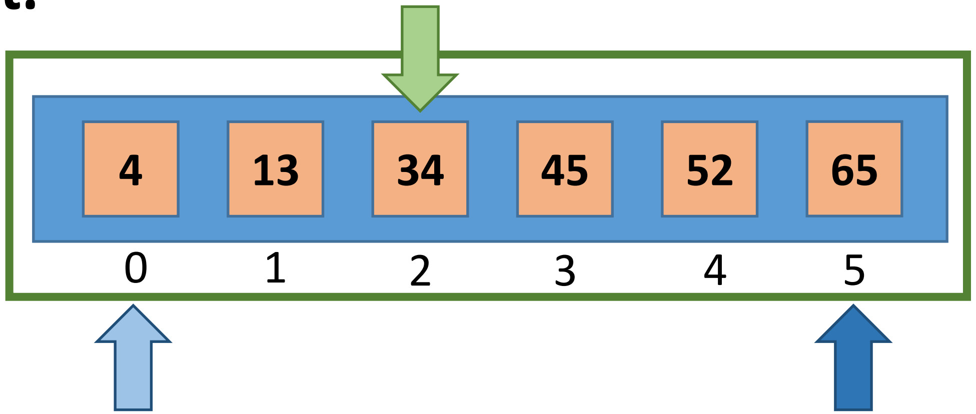
Binary Search (Searching for 12)

- The lower boundary is index 0.
- The upper boundary is the last index.
- To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: (0 + 5)/2 = 2.5 -> 2



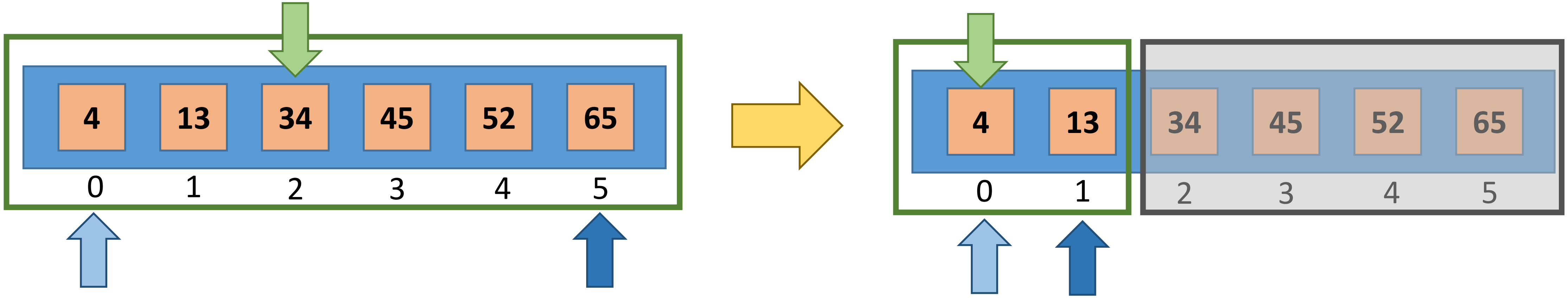
Binary Search (Searching for 12)

- Next, we do one of three things:
  - If the value we are seeking (12) is at this middle index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this list.
  - **If the value we are seeking is less than this value, then we will search the lower half of this list.**



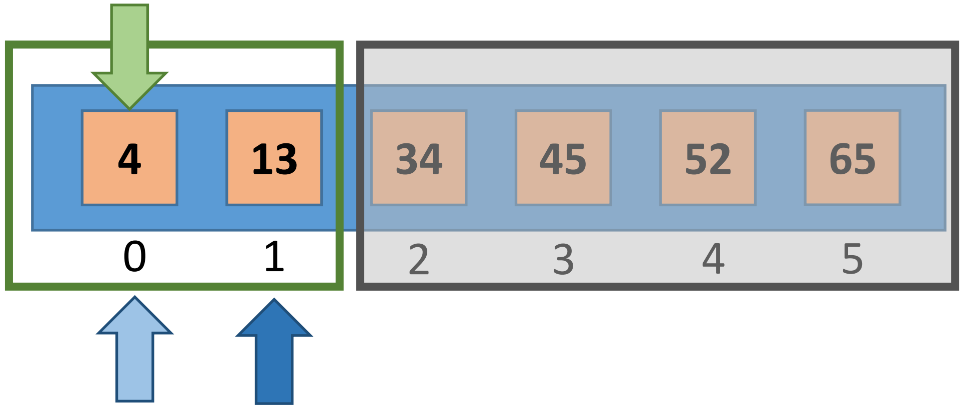
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Does not change.
  - Upper Boundary: Middle Index – 1 -> 2 – 1 = 1
  - Middle Index: (Lower + Upper) / 2 -> (0 + 1) /2 = 0.5 -> 0



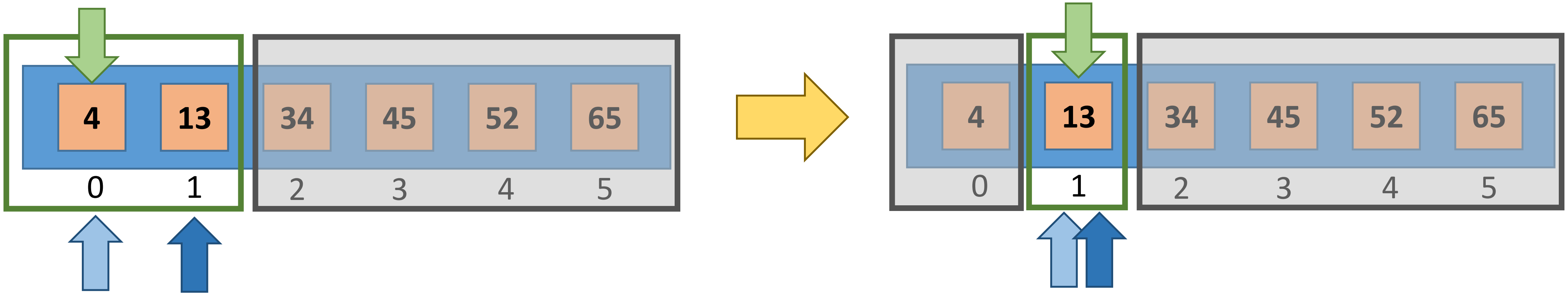
Binary Search (Searching for 12)

- We start the process over:
  - If the value we are seeking (12) is at this index, we are done searching.
  - **If the value we are seeking is greater than this value, then we will search the upper half of this list.**
  - If the value we are seeking is less than this value, then we will search the lower half of this list.



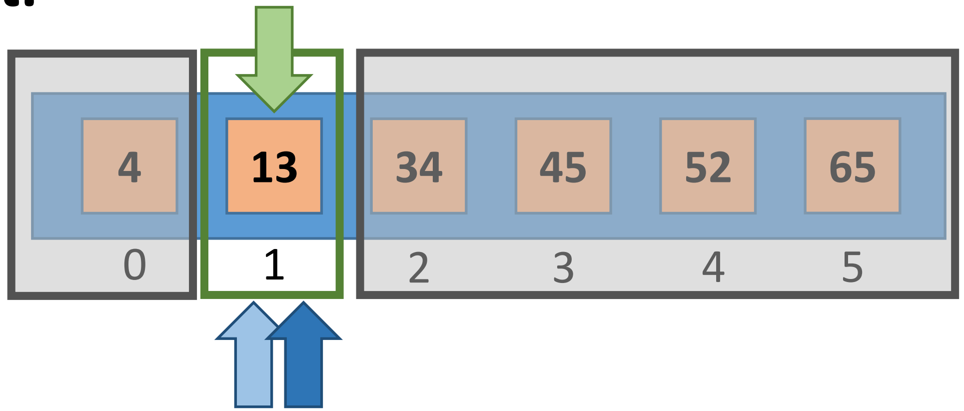
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Middle Index + 1 -> 0 + 1 = 1
  - Upper Boundary: Does not change.
  - Middle Index: (Lower + Upper) / 2 -> (1 + 1) /2 = 1



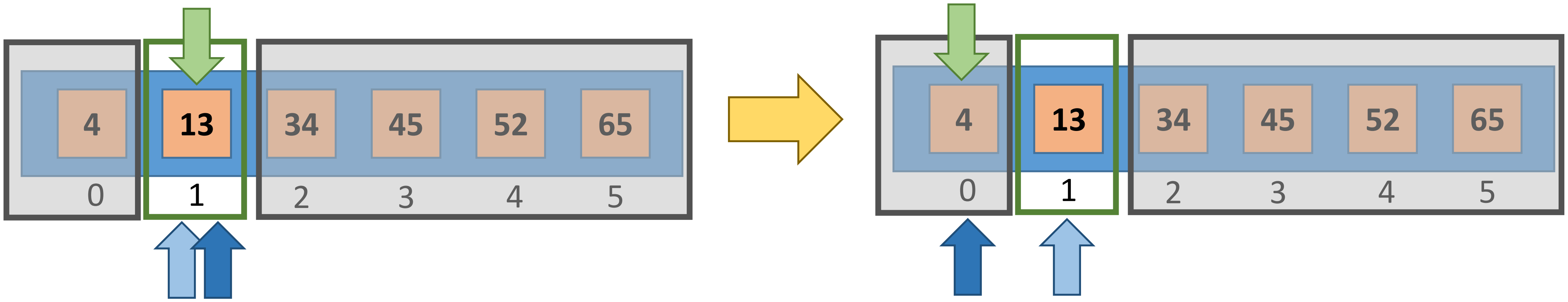
Binary Search (Searching for 12)

- We start the process over:
  - If the value we are seeking (12) is at this index, we are done searching.
  - If the value we are seeking is greater than this value, then we will search the upper half of this list.
  - **If the value we are seeking is less than this value, then we will search the lower half of this list.**



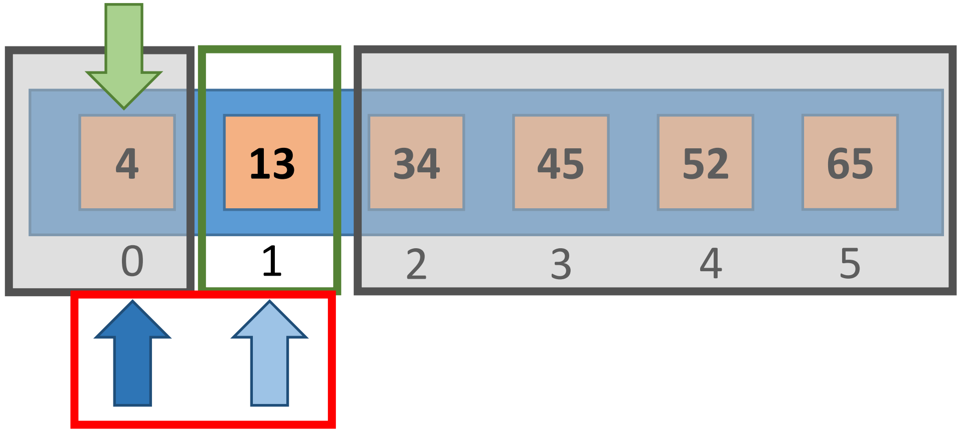
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
  - Lower Boundary: Does not change.
  - Upper Boundary: Middle Index – 1 -> 1 – 1 = 0
  - Middle Index: (Lower + Upper) / 2 -> (1 + 0) /2 = 0.5 -> 0



Binary Search (Searching for 12)

- When the upper boundary is less than the lower boundary, the algorithm will "give up."



Binary Search (Java code)

```java
int foundIndex = -1; //Will be set to the correct index when/if 45 is found
int lowBoundary = 0; //Low boundary index (Starts with index zero)
int highBoundary = array.length-1; //High boundary index (Starts with the last index)

while(highBoundary >= lowBoundary) { //Controls when the algorithm will "give up" and stop
    int middleIndex = (lowBoundary + highBoundary) / 2; //The index between the low and high bounds
    if (array[middleIndex] == 45) {
        foundIndex = middleIndex; //45 was found. Save the index and exit the loop
        break;
    }
    else if (45 > array[middleIndex]) {  //The value sought is greater than the middle value
        lowBoundary = middleIndex + 1;   //Set the new low boundary
    }
    else {                               //The value sought is less than the middle value
        highBoundary = middleIndex - 1;  //Set the new high boundary
    }
}
if(foundIndex == -1) {
    //If it still equals -1, then 45 was never found.
}
```

Binary Search

- The elements **must** be sorted (alphabetically, numerically, some order) or a binary search will not work.

- Best case scenario: The information sought is the middle element.

- Worst case scenario: You check (at most) half of the elements in an array or list.

Linear Search vs Binary Search

- Linear Searches do not require the array to be sorted.
  - Consider the array: {24, 74, 91, 13, 67, 45, 33, 89}
  - You could not use a binary search here because the array is not in order.

- Binary Searches will eliminate half of the possible values it needs to check after each iteration.
  - This doesn’t necessarily mean it is always faster, especially if you need to sort the array/list first.