Chapter 6 – Arrays, Part B – Searching for items in an Array

Linear Search

A common task in programming is that you need to search for something in an array. If there is no ordering on the values (*i.e.* they are not sorted), then we generally do a *linear search*. Suppose you have an array of test scores and you want to see if the test score 88 is in the array. The value that is sought is sometimes referred to as the *key*. This version of the linear search is very simple: iterate over the values in the array, testing each one to see if it equals the *key*. It is common for such a method to return the *index* of the position if the search was successful and -1 if the search was not successful. Below is a method that does a linear search for a *key*.

```
public static int linearSearch( int[] list, int key )
{
      for( int i=0; i<list.length; i++ )
      {
            if( key == list[i] )
                  return i;
      }
      return -1
}
```

Example 1

Of course, we can think of many variations of this algorithm. For instance, suppose that we want to find the index of the value that is closest to the key.

```
   public static void main(String[] args)
   {
      int[] scores = { 78, 84, 88, 56, 69, 92, 89, 97 };

      int key = 48;

      index = linearSearchClosest( scores, key );

      System.out.println( index + ", " + scores[index] );
   }

   public static int linearSearchClosest( int[] list, int key )
   {
      int distance, minDist = 1000, keyForMin = -1;

      for( int i=0; i<list.length; i++ )
      {
         distance = Math.abs( list[i] - key );

         if ( distance < minDist )
         {
            keyForMin = i;
            minDist = distance;
         }
      }
      return keyForMin;
   }
```

Example 2

Suppose that we want to find the index of the value that is closest to the key, but less than the key?

a. Analysis:

	0	1	2	3	4	5
vals→	56	73	38	51	33	47

Key	Index
46	2
58	0
22	-1
100	1

b. Algorithm:

```
Loop over list
        IF list[i] < key
                Calculate distance from key
                IF distance < minDistance
                        Update minDistance
                        Remember current index
Return index
```

c. Code:

```
public static int linearSearchLessThan( int[] list, int key )
{
   int distance, minDist = 1000, keyForMin = -1;

   for( int i=0; i<list.length; i++ )
   {
      if ( list[i] < key )
      {
         distance = key - list[i];

         if ( distance < minDist )
         {
            keyForMin = i;
            minDist = distance;
         }
      }
   }
   return keyForMin;
}
```

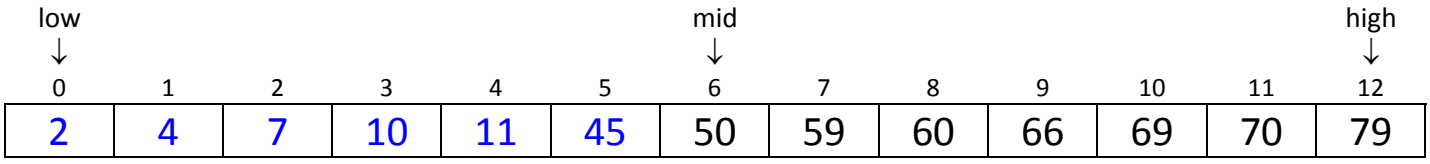
Binary Search, Idea

a. If the array is sorted (say, ascending), we can use a much faster algorithm, the *binary search.* The central idea is to successively cut the list in half until you find the value. You start by determining if the key is in the lower or upper half of the list. Suppose the key is in the lower half, then we discard the upper half of the list. Thus, the new list is the old lower half. We continue this process until we have found the key or detected its absence.

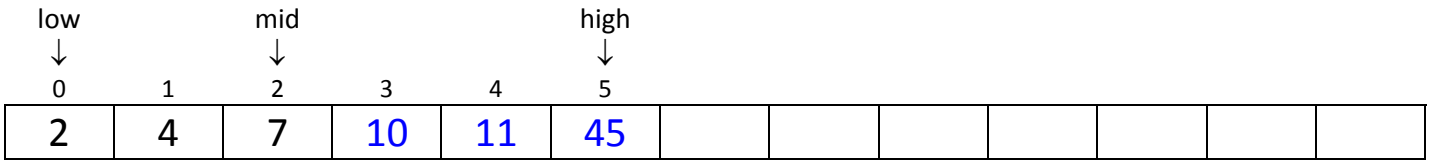
b. Binary Search, Idea, Example 1

Searching for: key = 11

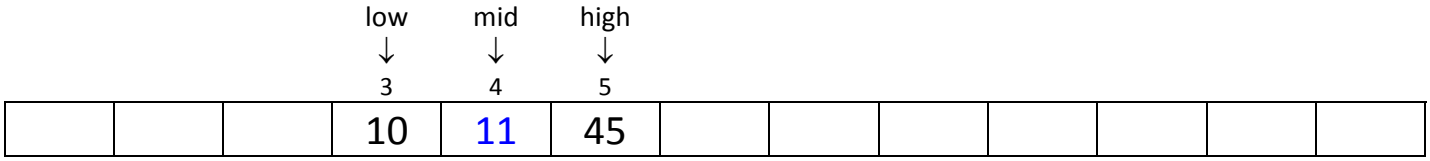
See if key is middle value, or is on right or left of middle value



See if key is middle value, or is on right or left of middle value



See if key is middle value, or is on right or left of middle value

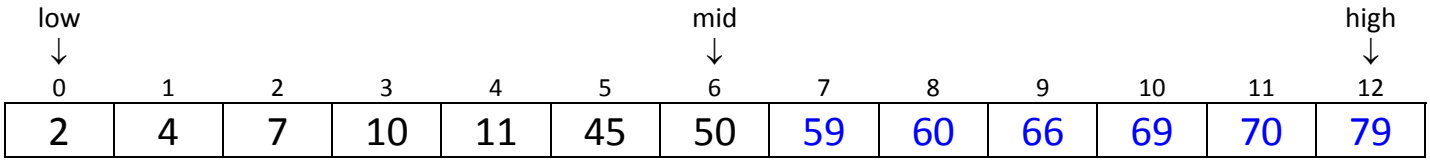


All done. *key* was found in position 4. It is customary for the binary search algorithm to return the index where the key was found.

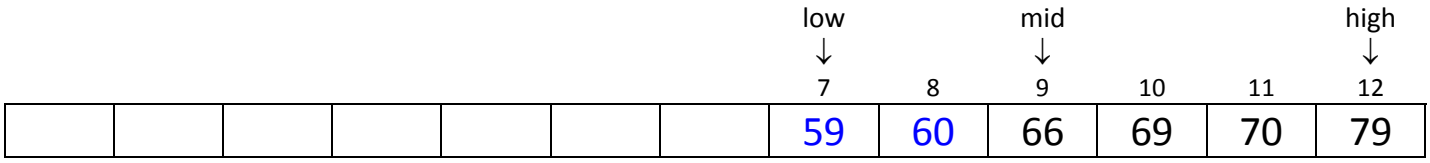
c. Binary Search, Idea, Example 2

Searching for: key = 62

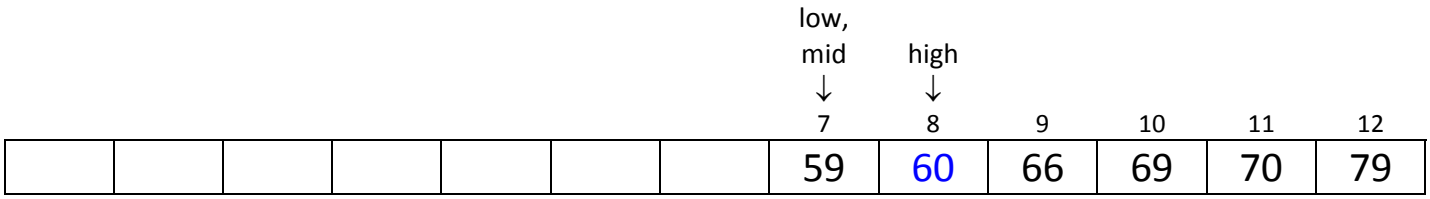
See if key is middle value, or is on right or left of middle value



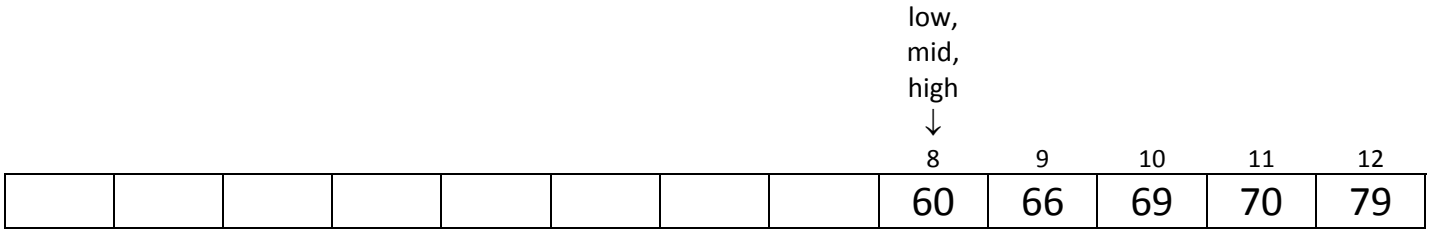
See if key is middle value, or is on right or left of middle value



See if key is middle value, or is on right or left of middle value



See if key is middle value, or is on right or left of middle value



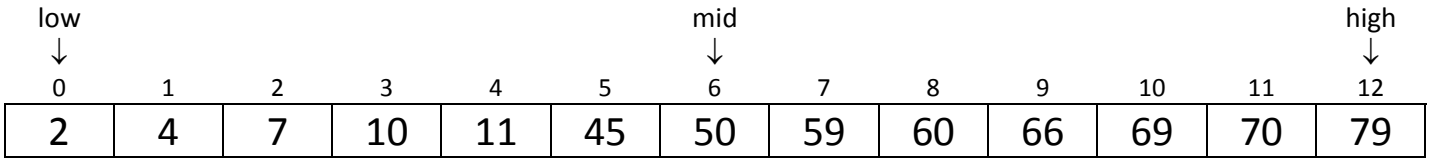
`low > high` means that the key fall between them. Thus, if the key were to be inserted in the proper order, it would be the 10th element (index 9). It is customary for the binary search algorithm to return -10 in this situation (for this example). The negative sign means that *the key was not found*. The value 10 means that if the key were to be inserted in the (sorted) list, it would be the 10th element and so would be located in the element with index=9, `list[9]`. This begs the question as to why the method algorithm doesn't just return -9. Consider the next example.

d. Binary Search, Idea, Example 3

Suppose that we are searching for the key = 2. What would the method return? Answer: 0

Now, suppose that we are searching for the key = 1. What would the method return?

See if key is middle value, or is on right or left of middle value



We see that *low* contains the index of the position where the key belongs, if the key was not found. So, we add 1 to the value of *low* and negate it to take care of the case when the key belongs in position 0. Thus, if binary search returned -1, this would mean that the key was smaller than any element in the list and if the key was inserted into the list, it belongs in the 1st element, with index 0.

Binary Search, Algorithm

a. First pass:

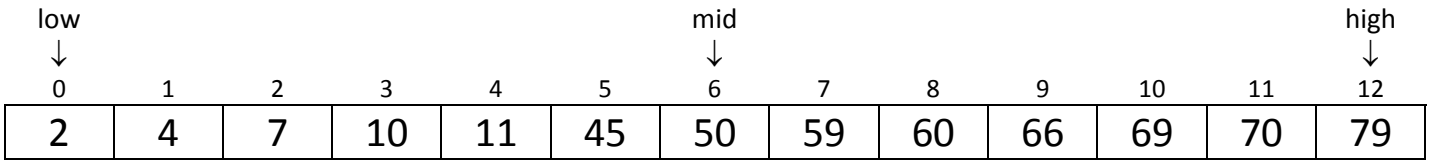
```
Initialize low=0, high=list.lenth-1
While low <= high
        Calculate mid=(low+high)/2
        IF key = mid
                Return mid
        ELSE IF key < mid
                high=mid-1
        ELSE
                low=mid+1
Return -(row+1)
```

b. Second pass:

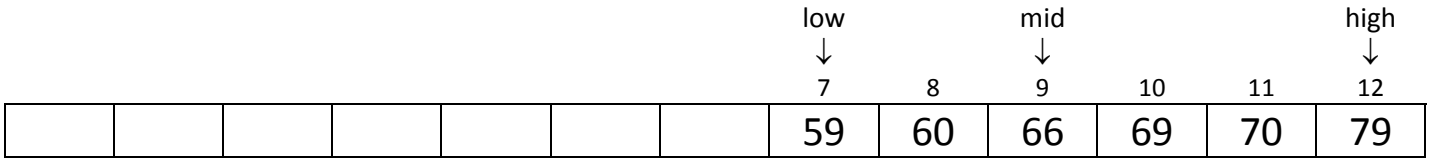
```
Initialize low=0, high=list.lenth-1
While low <= high
        Calculate mid=(low+high)/2
        IF key < list[mid]
                high=mid-1
        ELSE IF key > list[mid]
                low=mid+1
        ELSE
                Return mid
Return -(row+1)
```

Binary Search, Example 2

```
searching for: key = 62

initialize: low = 0; high = list.length-1

while ( high >= low )                          // true, 12 >= 0

      calculate: mid = (low+high)/2            // mid = (0+12)/2=6
```

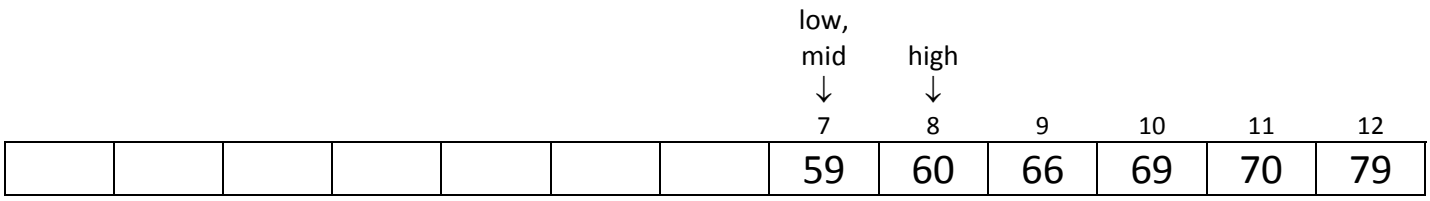



```
      if ( key < list[mid] )                   // false

            high = mid – 1;

      else if ( key > list[mid] )              // true, 62 > 50

            low = mid + 1                      // low = 6+1 = 7


while ( high >= low )                          // true, 12 >= 7

      calculate: mid = (low+high)/2            // mid = (7+12)/2 = 9
```




```
      if ( key < list[mid] )                   // true, 62 < 66

            high = mid – 1;                    // high = 9-1 = 8
```

```
while ( high >= low )                    // true, 8 >= 7
      calculate: mid = (low+high)/2      // mid = (7+8)/2 = 7
```




```
      if ( key < list[mid] )             // false
            high = mid – 1;
      else if ( key > list[mid] )        // true, 62 > 59
            low = mid + 1                // low = 7+1 = 8

while ( high >= low )                    // true, 8 >= 8
      calculate: mid = (low+high)/2      // mid = (8+8)/2 = 8
```


```
      if ( key < list[mid] )             // false
            high = mid – 1;
      else if ( key > list[mid] )        // true, 62 > 60
            low = mid + 1                // low = 8+1 = 9

while ( high >= low )                    // false, 8 >= 9
return –low-1                            // return -9-1 = -10
```

Thus, the binarySearch method returns -10. Again, the negative sign means that the key was not found. The value 10 means that if the key were to be inserted in the (sorted) list, it would be the 10th element and so would have be located in `list[9]`.

Example 3

a. Problem - Suppose that we want to modify the binary search algorithm so that it if the key is not found, it returns the position of the element that is closest to the key. For the example below, if we were searching for key=62, the binary search will return -10 as we saw before. With this new algorithm, we would like it to return -9, since list[8]=60 is closer to 62 than list[9]=66.

low ↓						mid ↓						high ↓
0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	7	10	11	45	50	59	60	66	69	70	79

b. Analysis:

The binary search algorithm will not change, except at the very end. Remember that *low*>*high* if the item is not found and that the key is between *high* and *low*:

list[high] < key < list[low]

Thus, we just need to calculate the distance between key and the two bounds and choose which one is closer.

c. Algorithm:

```
Initialize low=0, high=list.lenth-1
While low <= high
        Calculate mid=(low+high)/2
        IF key < list[mid]
                high=mid-1
        ELSE IF key > list[mid]
                low=mid+1
        ELSE
                Return mid
IF list[low] is closer to key
        Return –(row+1)
ELSE
        Return –(high+1)
```

d. Implementation – We would implement the last part this way:

```
int dLow = list[low] – key;
int dHigh = key - list[high];

if ( dLow < dHigh )
      return –low-1;
else
      return –high-1;
```