COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

Brief Review

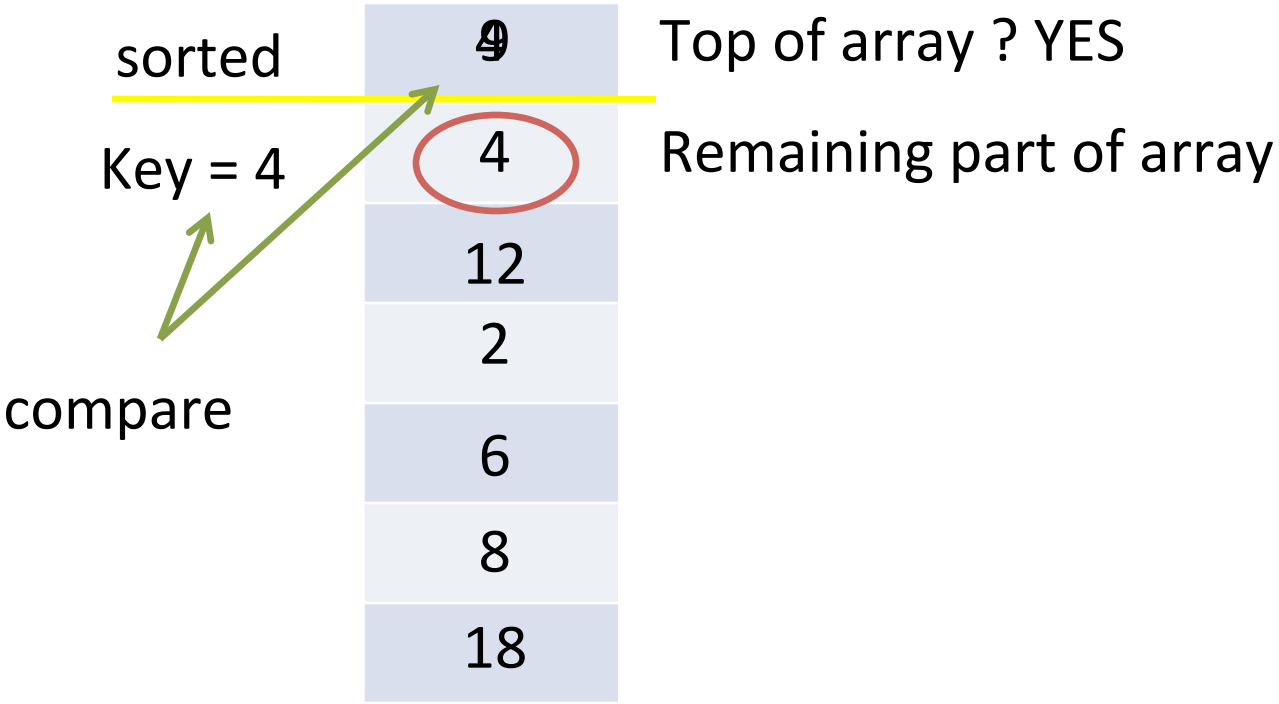
- Introduction to sorting algorithms
  - Selection sort
  - Insertion sort
- Implementing generic sort methods
- Today:
  - Wrap up insertion sort
  - Introduce searching algorithms

Review: Insertion Sort

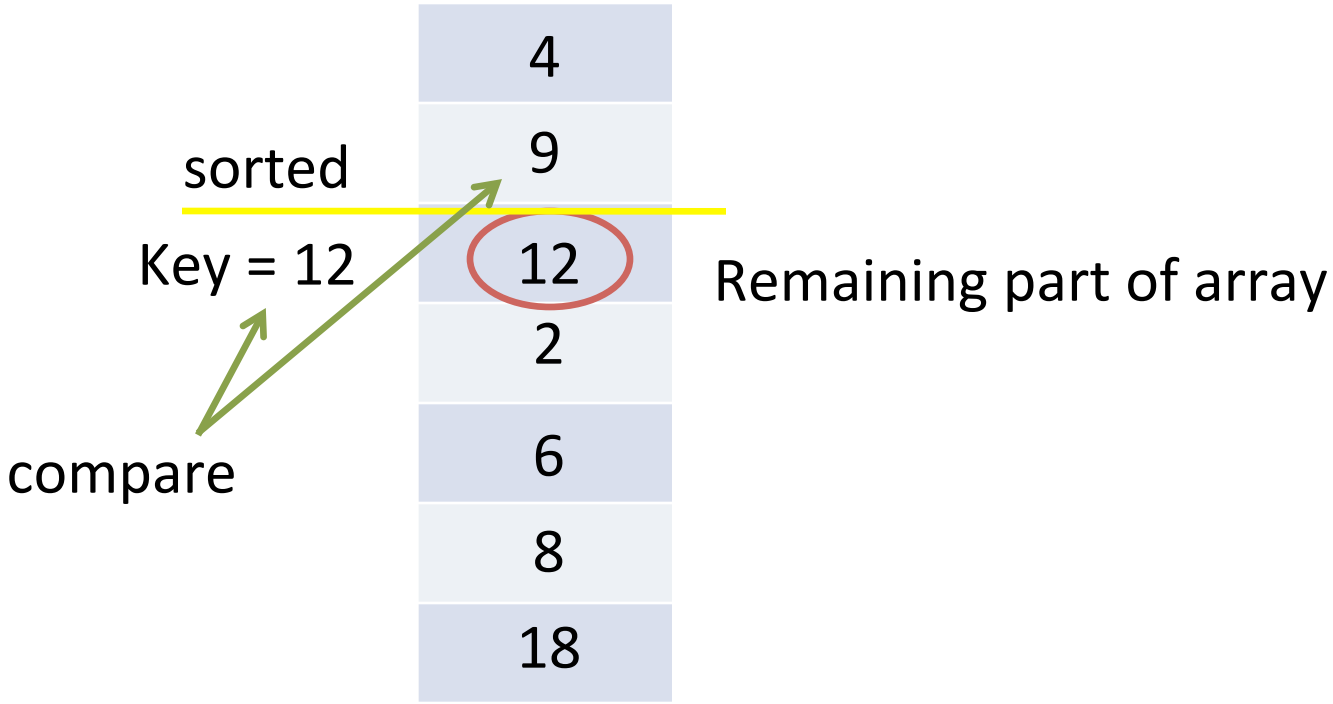
- Summary strategy of Insertion Sort:
  - Keep a sorted sublist and an unsorted sublist
  - Take an item from the unsorted sublist
  - Put it "in order" into the sorted side
  - Repeat until everything is sorted

- Makes use of in-place memory

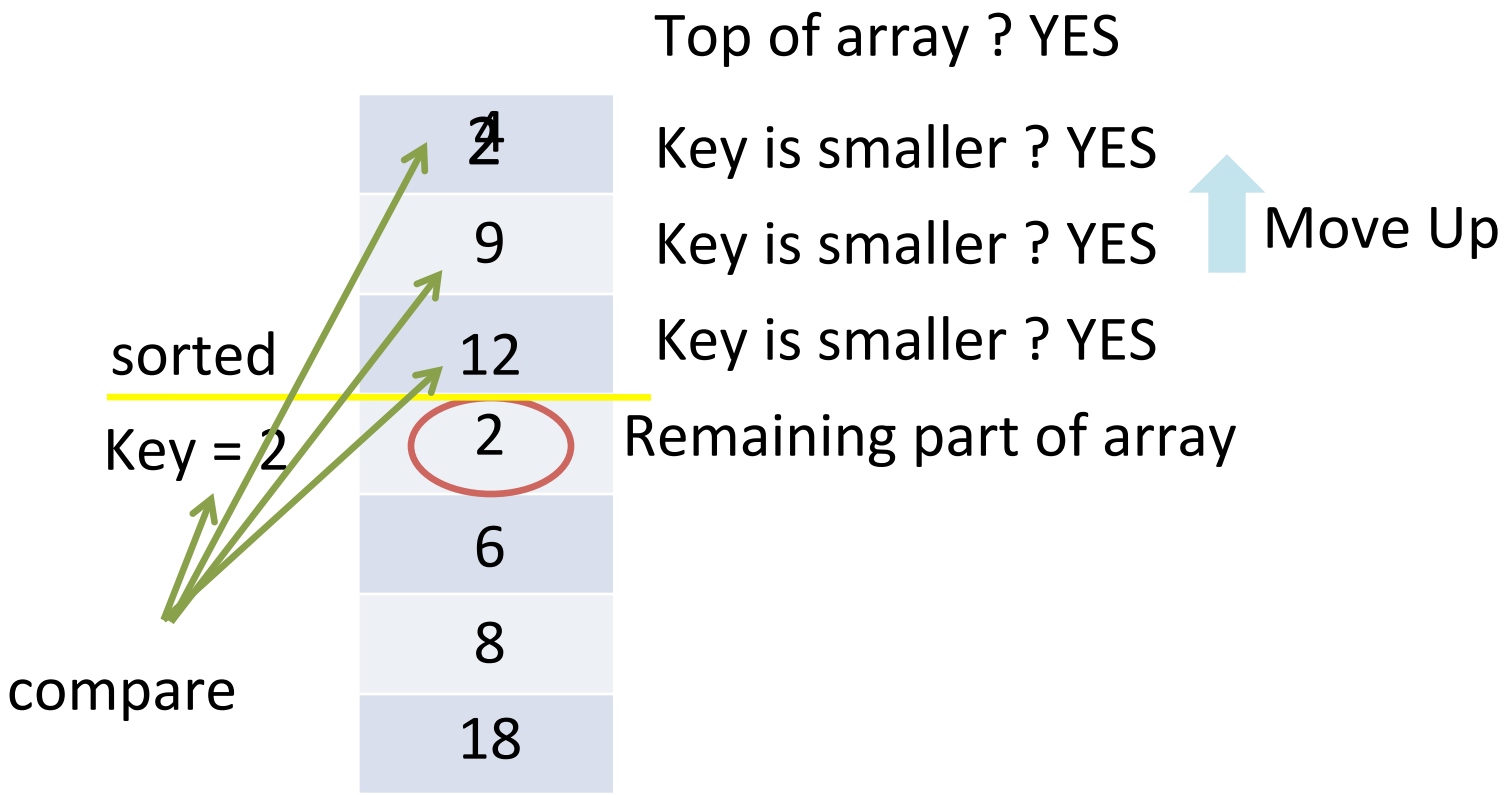
Review: Insertion sort – Round 1



Review: Insertion sort – Round 2



Review: Insertion sort – Round 3



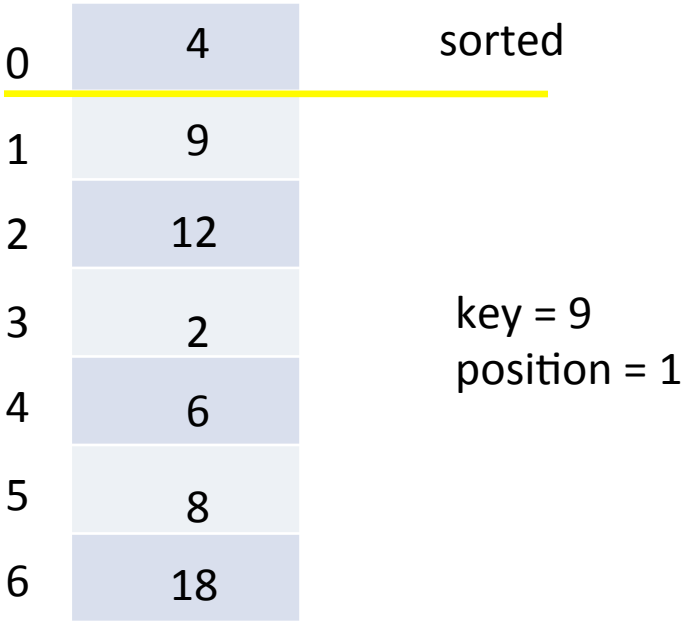
Defining Insertion Sort

```
public static void insertionSort( int[] list )
{
  //  1. created sorted sublist: item one is sorted
  //  2. for each unsorted element
  //     a. compare key with those in sorted sublist
  //     b. keep shifting larger values unless we reach top
  //     c. insert key when correct spot is found
}
```

```
public static void insertionSort( int[] list )
{
  //  1. created sorted sublist: item one is sorted
  int sorted = 0;
  //  2. for each unsorted element
  for( int i=(sorted+1); i<list.length; i++ )
  {
    //  a. compare key with those in sorted sublist
    //  b. keep shifting larger values unless we reach top
    //  c. insert key when correct spot is found
  }
}
```

```
public static void insertionSort( int[] list )
{
  //  1. created sorted sublist: item one is sorted
  int sorted = 0;
  //  2. for each unsorted element
  for( int i=(sorted+1); i<list.length; i++ )
  {
    //  a. compare key with those in sorted sublist
    int key = list[i];

    //  b. keep shifting larger values unless we reach top
    int position = i;
    while( position > 0 && list[position-1] > key )
    {
      // …
      position--;
    }

    //  c. insert key when correct spot is found
    list[position] = key;
  }
}
```

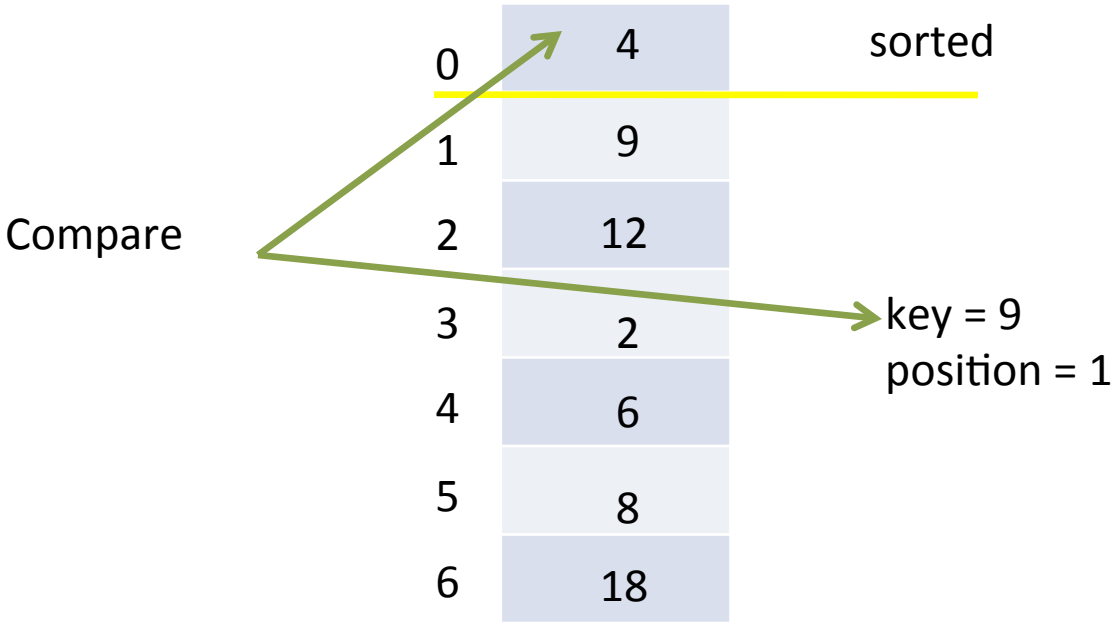
```
public static void insertionSort( int[] list )
{
  //  1. created sorted sublist: item one is sorted
  int sorted = 0;
  //  2. for each unsorted element
  for( int i=(sorted+1); i<list.length; i++ )
  {
    //  a. compare key with those in sorted sublist
    int key = list[i];

    //  b. keep shifting larger values unless we reach top
    int position = i;
    while( position > 0 && list[position-1] > key )
    {
      list[position] = list[position-1]; // shift
      position--;
    }

    //  c. insert key when correct spot is found
    list[position] = key;
  }
}
```




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
```


For all elements in the array

```
}
```

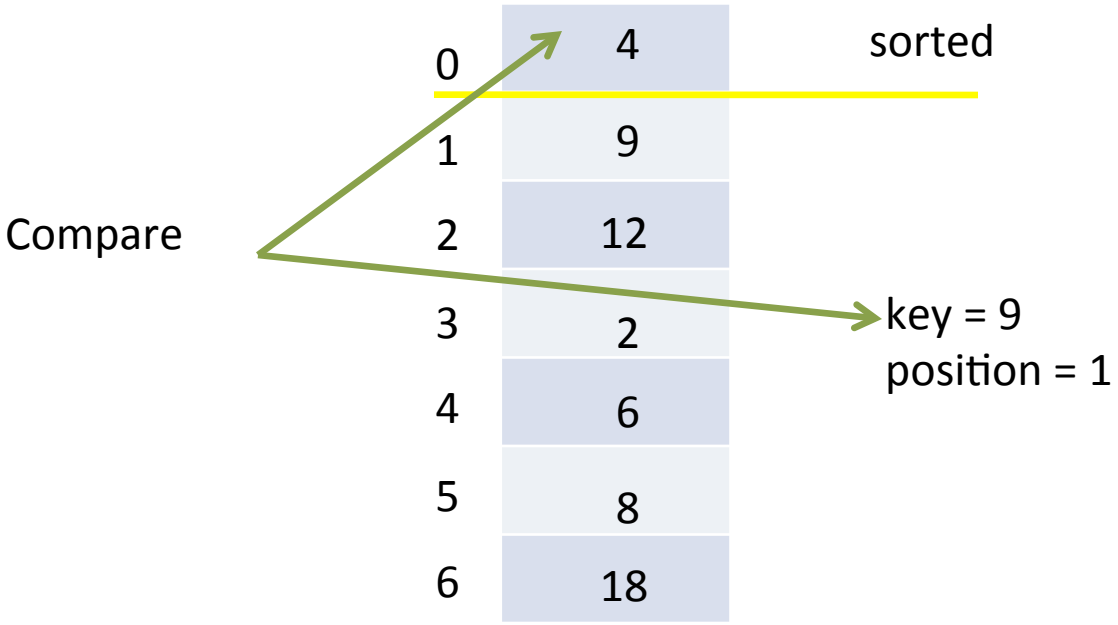



```
for (int i=1; index < list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

true && (4 > 9) ?

For all elements in the array

Keep shifting sorted elements as necessary



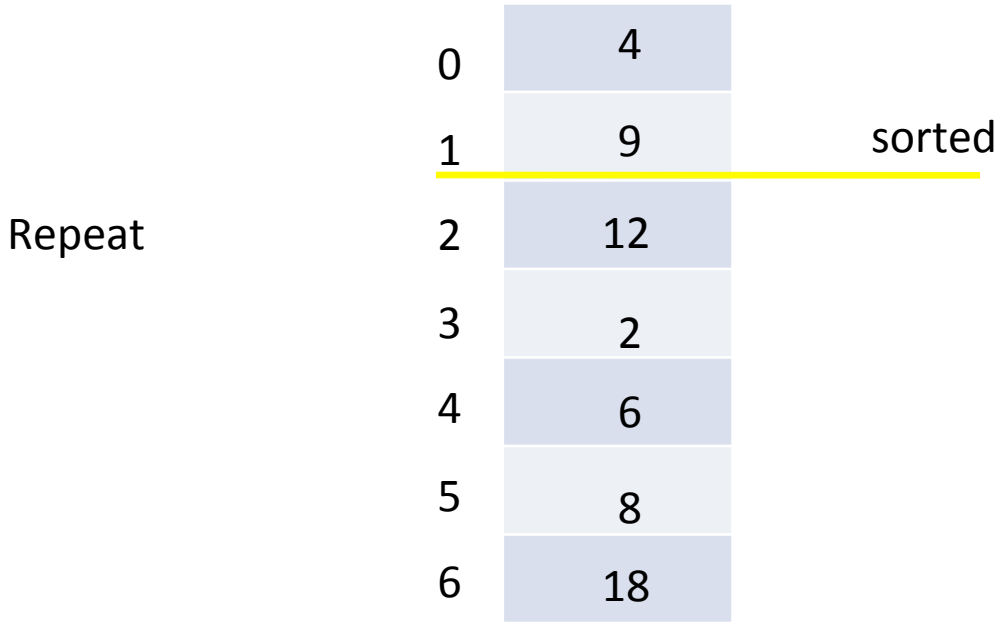

```
for (int i=1; index < list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

       list[position] = key;
}
```

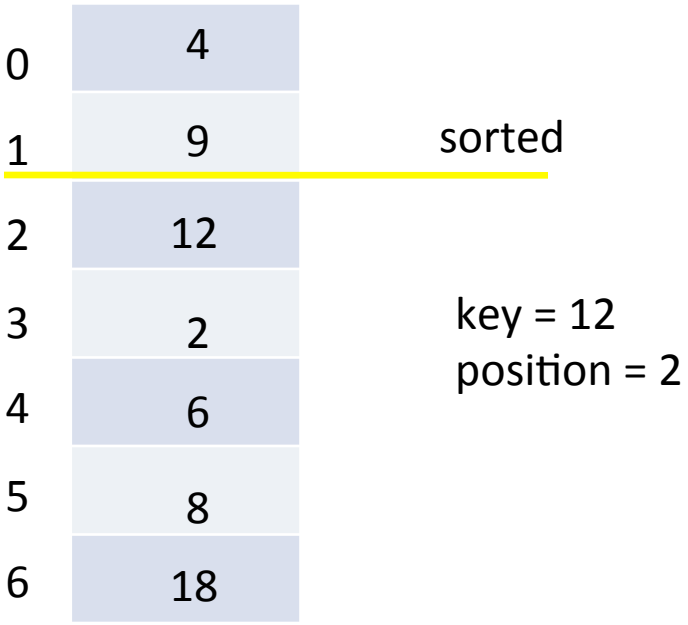
true && (4 > 9) ?

For all elements in the array

Keep shifting sorted elements as necessary

Insert key

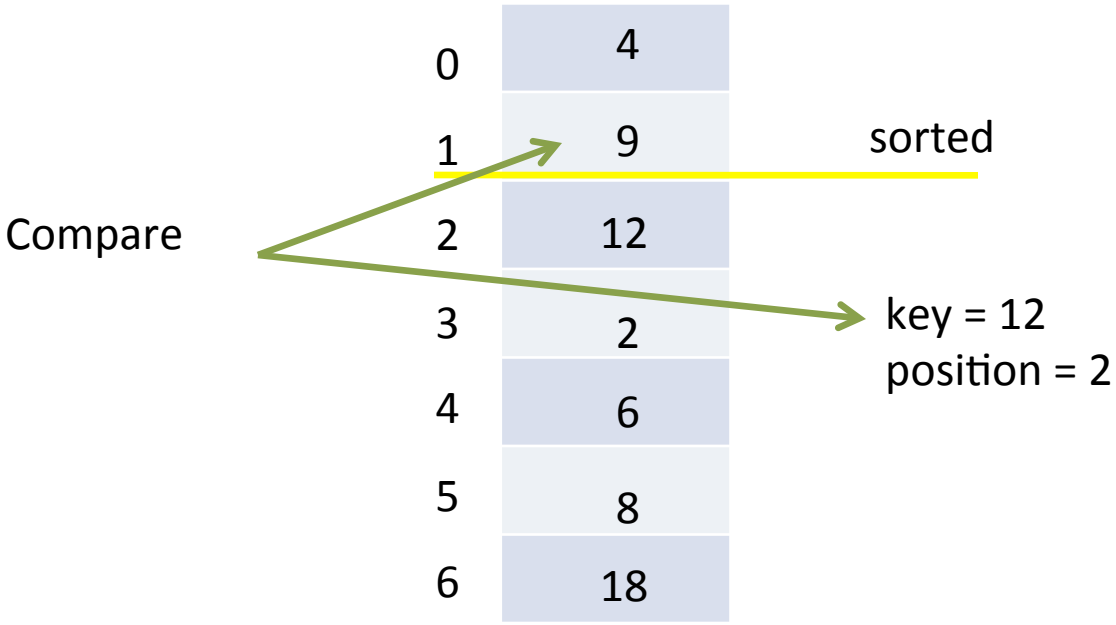

0
4
1
9
sorted
Repeat
2
12
3
2
4
6
5
8
6
18




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
```

For all elements in the array

```
}
```

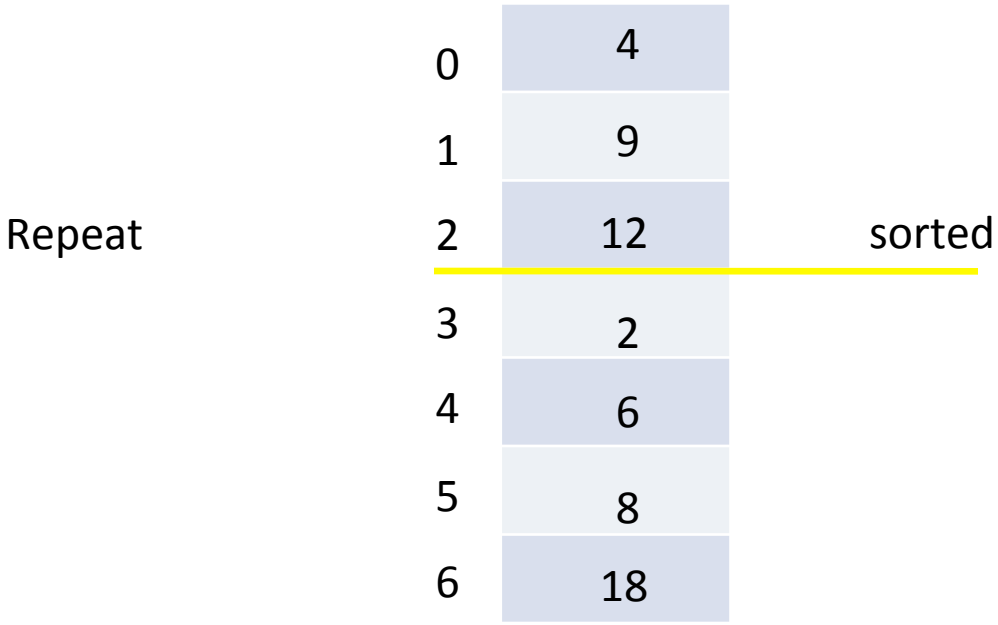



```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

true && (9 > 12) ?

For all elements in the array

Keep shifting sorted elements as necessary

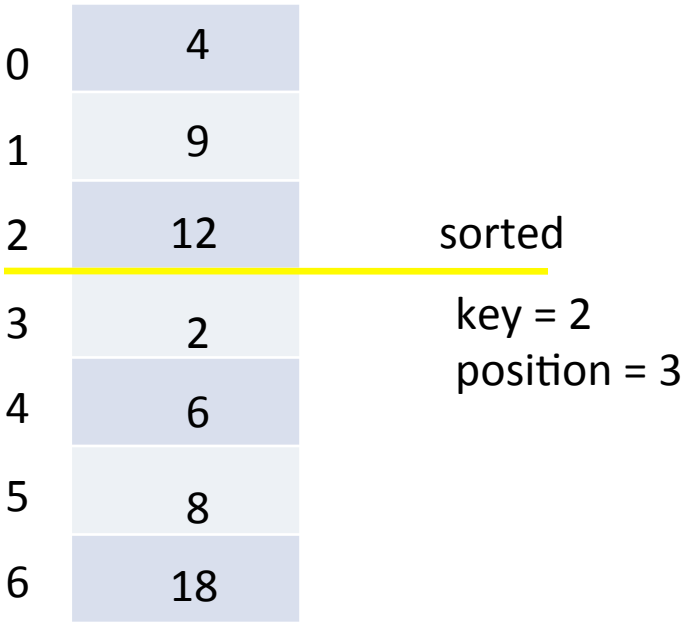



```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

       list[position] = key;
}
```

For all elements in the array

Keep shifting sorted elements as necessary

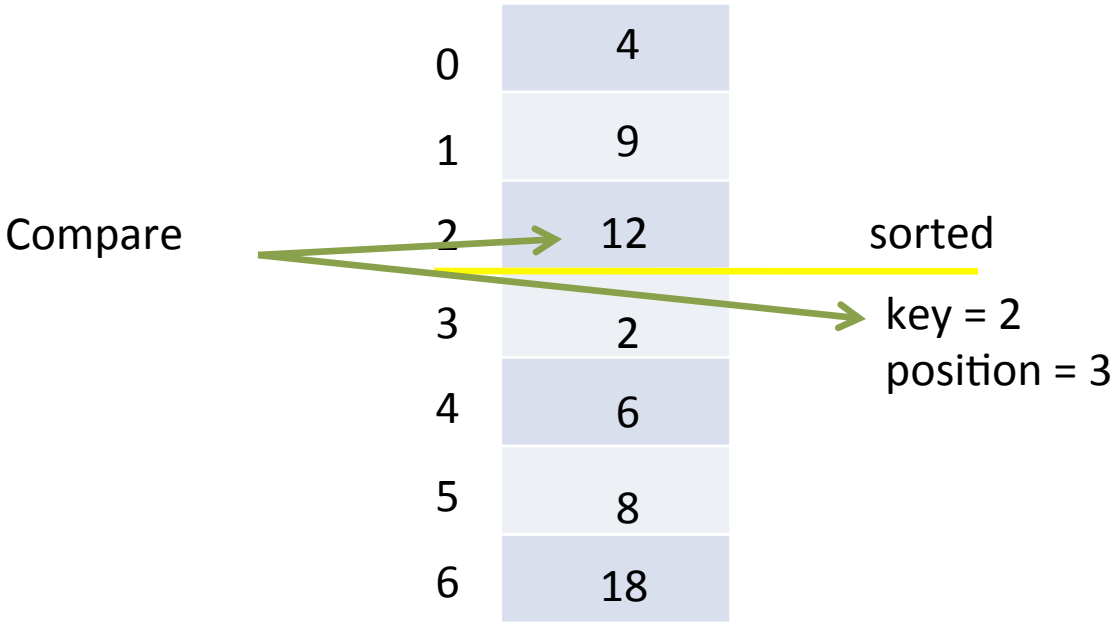
Insert key




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
```

For all elements in the array

```
}
```

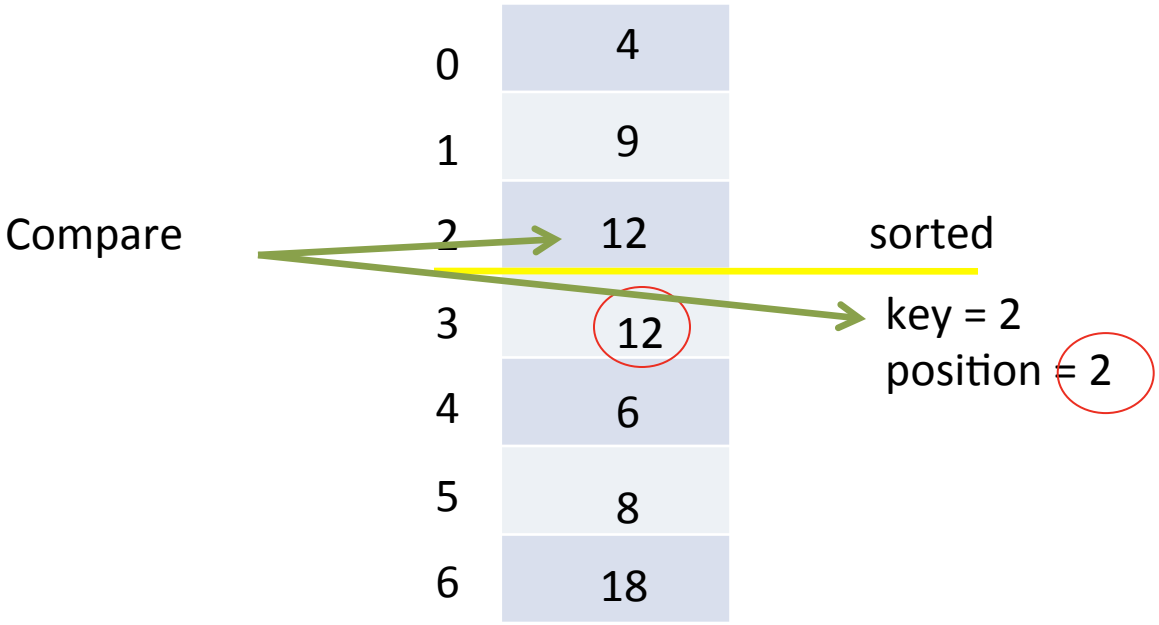



```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

true && (12 > 2) ?

For all elements in the array

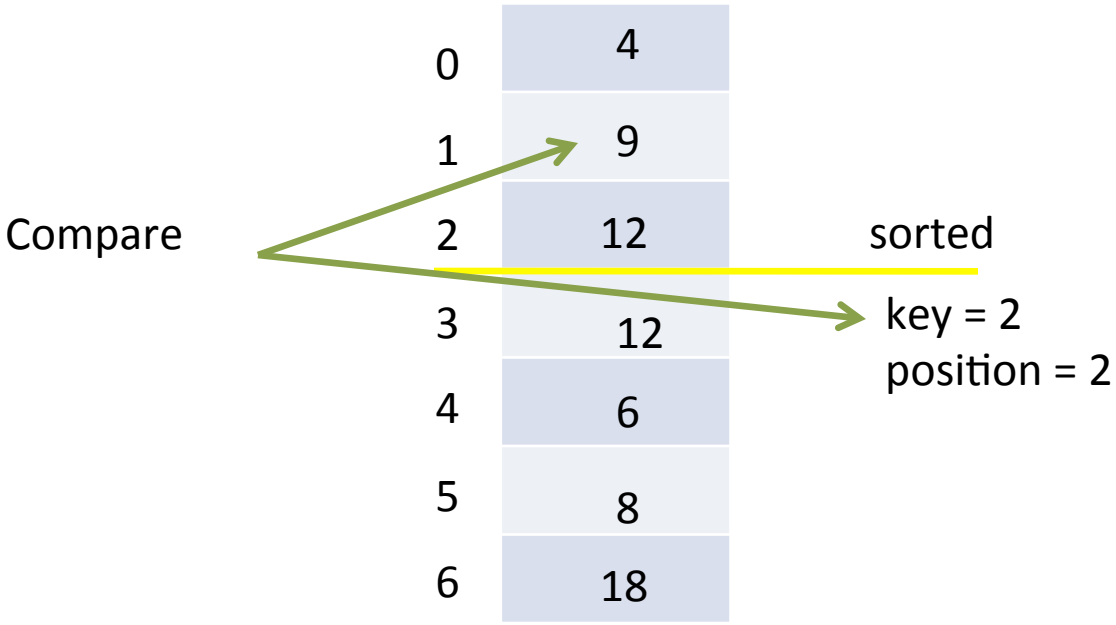
Keep shifting sorted elements as necessary




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

For all elements in the array

Keep shifting sorted elements as necessary

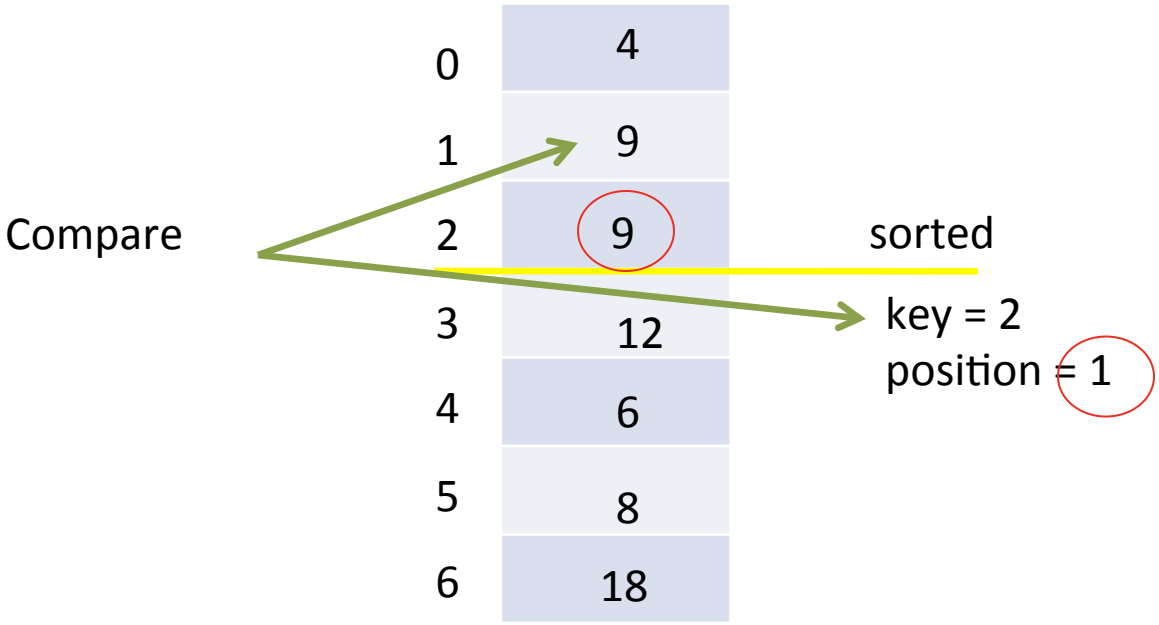



```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

true && (9 > 2) ?

For all elements in the array

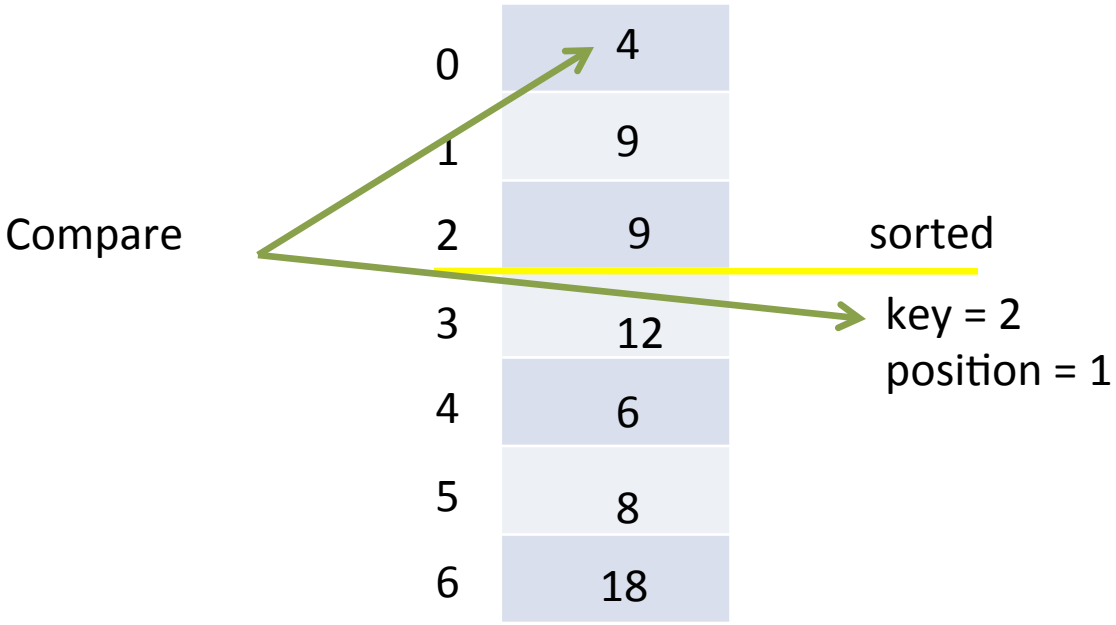
Keep shifting sorted elements as necessary




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

For all elements in the array

Keep shifting sorted elements as necessary

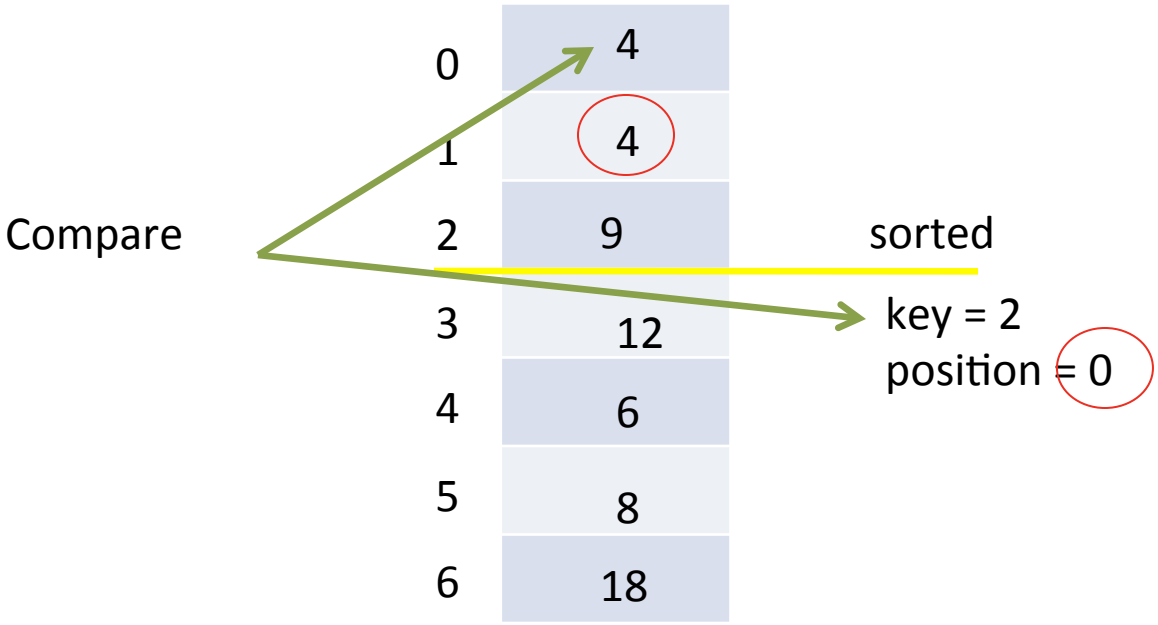



```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

true && (4 > 2) ?

For all elements in the array

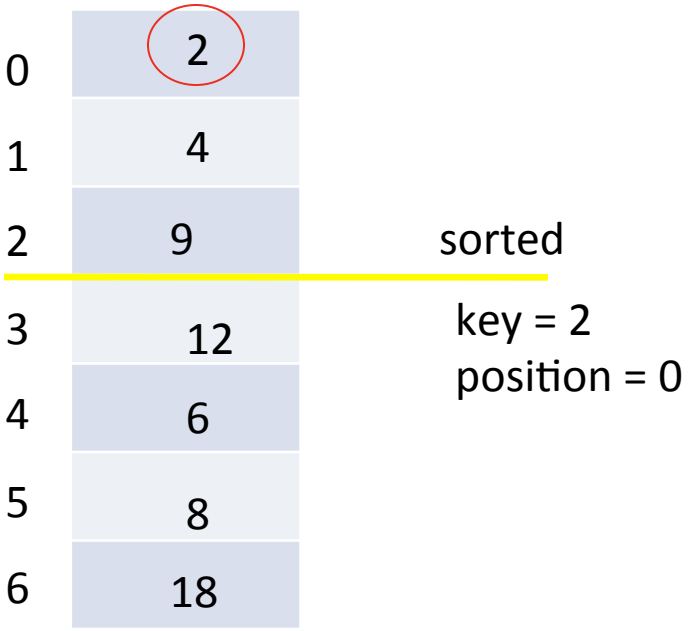
Keep shifting sorted elements as necessary




```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

}
```

For all elements in the array

Keep shifting sorted elements as necessary



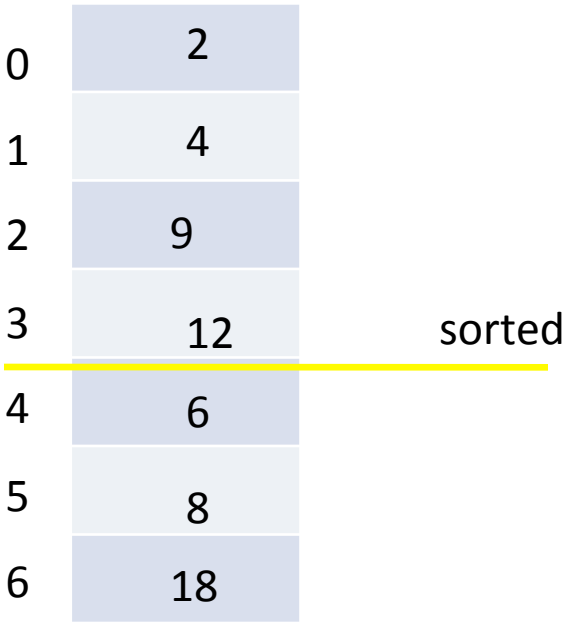

```
for (int i=1; i< list.length; i++) {
        int key = list[i];
        int position = i;
        while (position > 0 && list[position-1] > key) {
                list[position] = list[position – 1];
                position --;
         }

       list[position] = key;
}
```

For all elements in the array

Keep shifting sorted elements as necessary

Insert key

Repeat



Comparing Sorts

- Both selection and insertion sort have similar time efficiency
- Selection sort recap:
  - Outer loop scans all elements
  - Inner loop finds smallest in remaining elements
  - Swap
- Insertion sort recap:
  - Outer loop scans all elements
  - Inner loop finds correct spot in sorted sublist
  - Shift down

Comparing Sorts

- Both selection and insertion sort have similar time efficiency
- General Time Analysis for N elements
  - Outer loop scans all elements => N steps
  - Inner loop finds smallest in remaining elements
    => N-1 steps
  - Swap or Shift down => 1 step (independent of N)
- Loops are nested
  - For each of N steps, we do N-1 steps
    => N x (N-1) = N^2 steps
- We say these algorithms run in "Order of N^2" time

Bookshelf Scenario

- Searching is very closely related to sorting
- You have to put away a book
  - Where on the shelf do you put it?
  - What order?

Image from:
http://beattiesbookblog.blogspot.ca/

Bookshelf Scenario

Image from:
http://beattiesbookblog.blogspot.ca/

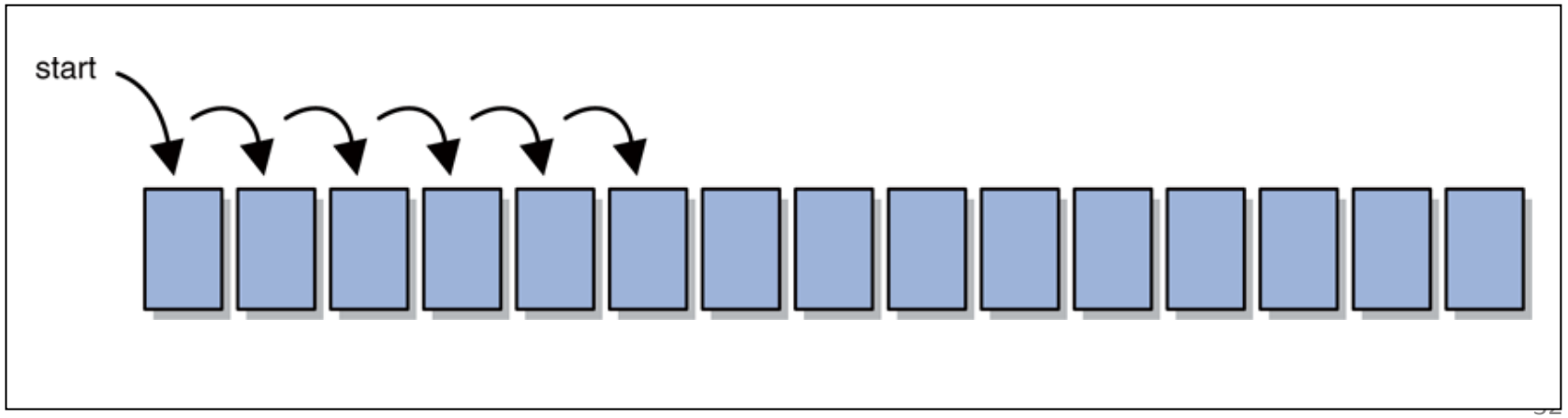
- Searching is very closely related to sorting
- You have to put away a book
  - Where on the shelf do you put it?
  - What order?
- Fast solution:
  - Put it in the first empty spot
  - Retrieval later will be difficult (especially if the shelf is full)
    e.g., search book by book
- Slower solution:
  - Put it in some meaningful order
    e.g., sort by author last name
  - Retrieval later will be fast (even if the shelf is full)
    e.g., "jump" to locations closer to where actual book may be

Searching

- Process of finding a target within a group of items called the search pool
  - Target may or may not be present!
- Want to perform search efficiently
  - Minimize number of comparisons
- Two classic algorithms:
  - Linear search
  - Binary search

Linear Search

- Steps:
  - Begins at one end of a list
  - Examines each element in turn
  - Eventually, either find target or reach end of list
- We did this in sorting (where?)



Example of Linear Search

- Search for a target in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37
  26 93 3 63 5

Example of Linear Search

- Search for a target in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37
  26 93 3 63 5

- E.g.: Target = 6
  - Compare with 47: no match
  - Compare with 74: no match
  - Compare with 6: match found

Example of Linear Search

- Search for a target in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37
  26 93 3 63 5

- E.g.: Target = 93
  - Works the same, but requires 18 comparisons

Example of Linear Search

- Search for a target in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37
  26 93 3 63 5

- E.g.: Target = 2
  - Works the same, but requires 21 comparisons and we get no match found

Implementing Linear Search

- Steps?

- How to write in Java code?

Implementing Linear Search

- Steps:
  - Check each element in the list
    - If found, return item
    - If not found, continue searching or exhaust the list
  - Otherwise, return not found or default value

- How to write in Java code?

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)

{
   // check each element in the list
   // if element matches target, then "remember" this item
   // return item
}
```

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{

   // check each element in the list
   while (. . .)
   {
      // if element matches target, then "remember" this item

   }

   // return item

}
```

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{

   // check each element in the list
   while ( . . . )
   {
      // if element matches target, then "remember" this item
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   // return item

}
```

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{

   // check each element in the list
   while ( . . . )
   {
      // if element matches target, then "remember" this item
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   // return item

}
```

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{
   int index = 0;
   boolean found = false;

   // check each element in the list
   while (!found && index < list.length)
   {
      // if element matches target, then "remember" this item
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   // return item

}
```

The `linearSearch` method in the `Searching` class:

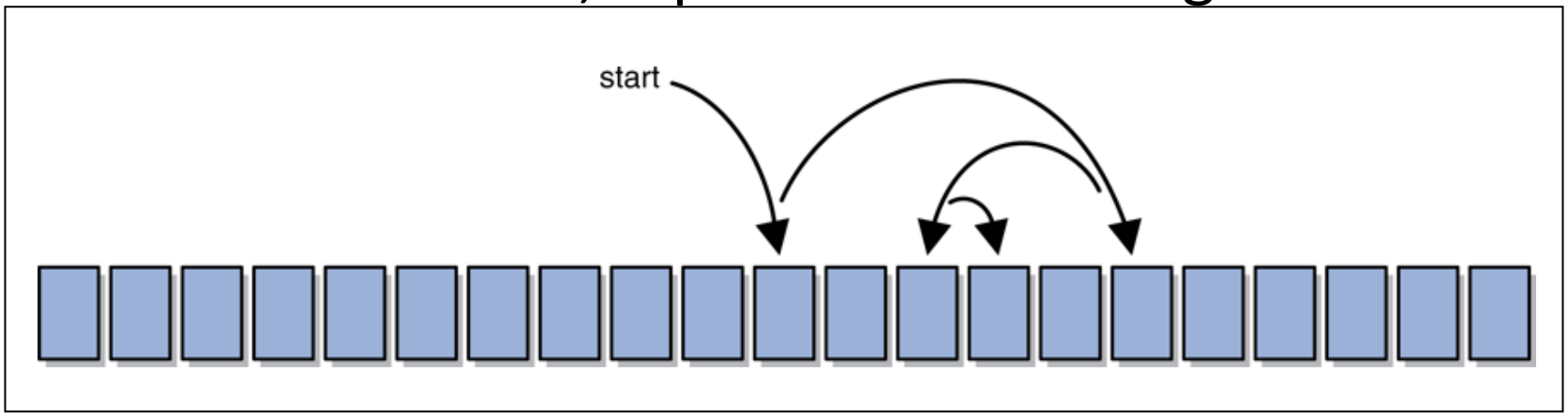
```
//-----------------------------------------------------------------
//  Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{
   int index = 0;
   boolean found = false;

   // check each element in the list
   while (!found && index < list.length)
   {
      // if element matches target, then "remember" this item
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   // return item
   if (found)
      return list[index];
   else
      return null;
}
```

Binary Search

- Assumes list of items in search pool is sorted
- Steps:
  - Examine middle element
  - If item matches target, search is over
  - If item is larger, repeat search on smaller half
  - If item is smaller, repeat search on larger half
- A single comparison eliminates a large part of search pool

Binary Search

- Assumes list of items in search pool is sorted
- Steps:
  - Examine middle element
  - If item matches target, search is over
  - If item is larger, repeat search on smaller half
  - If item is smaller, repeat search on larger half



Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37
  26 93 3 63 5

Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37 26 93 3 63 5

- Target = 97
  - Compare with 75:

Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37 26 93 3 63 5

- Target = 97
  - Compare with 75: target > 75 so check right half
    - Focus only on right half: 83 97 94 22 37 26 93 3 63 5
  - Compare with 37:

Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37 26 93 3 63 5

- Target = 97
  - Compare with 75: target > 75 so check right half
    - Focus only on right half: 83 97 94 22 37 26 93 3 63 5
  - Compare with 37: target > 37 so check right half
    - Focus only on right half: 26 93 3 63 5
  - Compare with 3:

Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37 26 93 3 63 5

- Target = 97
  - Compare with 75: target > 75 so check right half
    - Focus only on right half: 83 97 94 22 37 26 93 3 63 5
  - Compare with 37: target > 37 so check right half
    - Focus only on right half: 26 93 3 63 5
  - Compare with 3: target > 3 so check right half
    - Focus only on right half: 63 5
  - Compare with 5:

Example of Binary Search on Unsorted List

- Search for 97 in:
  47 74 6 85 28 88 7 21 20 70 75 83 97 94 22 37 26 93 3 63 5

- Target = 97
  - Compare with 75: target > 75 so check right half
    - Focus only on right half: 83 97 94 22 37 26 93 3 63 5
  - Compare with 37: target > 37 so check right half
    - Focus only on right half: 26 93 3 63 5
  - Compare with 3: target > 3 so check right half
    - Focus only on right half: 63 5
  - Compare with 5: target > 5, nothing left in right half
  - Return not found

- This is why a sorted list is required

Example of Binary Search on Sorted List

- Sort list first and then search for 97 in:
  3 5 6 7 21 20 22 26 28 37 47 63 70 74 75 83 85 88 93 94 97

- Target = 97
  - Compare with 47:

Example of Binary Search on Sorted List

- Sort list first and then search for 97 in:
  3 5 6 7 21 20 22 26 28 37 47 63 70 74 75 83 85 88 93 94 97

- Target = 97
  - Compare with 47: target > 47 so check right half
    - Focus only on right half: 63 70 74 75 83 85 88 93 94 97
  - Compare with 83:

Example of Binary Search on Sorted List

- Sort list first and then search for 97 in:
  3 5 6 7 21 20 22 26 28 37 47 63 70 74 75 83 85 88 93 94 97

- Target = 97
  - Compare with 47: target > 47 so check right half
    - Focus only on right half: 63 70 74 75 83 85 88 93 94 97
  - Compare with 83: target > 83 so check right half
    - Focus only on right half: 85 88 93 94 97
  - Compare with 93:

Example of Binary Search on Sorted List

- Sort list first and then search for 97 in:
  3 5 6 7 21 20 22 26 28 37 47 63 70 74 75 83 85 88 93 94 97

- Target = 97
  - Compare with 47: target > 47 so check right half
    - Focus only on right half: 63 70 74 75 83 85 88 93 94 97
  - Compare with 83: target > 83 so check right half
    - Focus only on right half: 85 88 93 94 97
  - Compare with 93: target > 93 so check right half
    - Focus only on right half: 94 97
  - Compare with 97: match found

Implementing Binary Search

- Steps?

- How to write in Java code?

Implementing Binary Search

- Steps:
  - Keep searching until found or nothing's left in list
    - Compare against middle element
    - Repeat on smaller or larger half if no match
  - Return item
- What do we have to keep track of?
- How to write in Java code?

Implementing Binary Search

- Intermediate output:

```
Arr2: 0, 2, 4, 6, 7, 8, 9, 11, 12, 13, 18, 22, 26, 32, 39
Testing binary search on sorted list...
min=0,max=15, mid=7
min=8,max=15, mid=11
min=8,max=10, mid=9
min=10,max=10, mid=10
18
min=0,max=15, mid=7
min=8,max=15, mid=11
min=8,max=10, mid=9
min=10,max=10, mid=10
-1
```

- How to write in Java code?

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{

   // keep searching until found or when nothing left in the pool

      // check middle element

         // if no match: decide on which half to repeat search on

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element

         // if no match: decide on which half to repeat search on

   }

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         // if no match: decide on which half to repeat search on

   }

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         // if no match: decide on which half to repeat search on

   }

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         // if no match: decide on which half to repeat search on
         if (target.compareTo(list[mid]) < 0)
            max = mid-1;
         else
            min = mid+1;
   }

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         // if no match: decide on which half to repeat search on
         if (target.compareTo(list[mid]) < 0)
            max = mid-1;
         else
            min = mid+1;
   }

   // return result

}
```

The `binarySearch` method in the `Searching` class:

```
//  Assumes list is already sorted in ascending order when passed in
//  Returns a reference to target object if found, or returns null
public static Comparable binarySearch (Comparable[] list, Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   // keep searching until found or when nothing left in the pool
   while (!found && min <= max)
   {
      // check middle element
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         // if no match: decide on which half to repeat search on
         if (target.compareTo(list[mid]) < 0)
            max = mid-1;
         else
            min = mid+1;
   }

   // return result
   if (found)
      return list[mid];
   else
      return null;
}
```

Comparing Search Algorithms

- Linear search
  - Easy to understand and implement
  - No expectations on input
  - Order N, given N items to search through

Comparing Search Algorithms

- Linear search
  - Easy to understand and implement
  - No expectations on input
  - Order N, given N items to search through
- Binary search
  - Easy to understand, harder to implement
  - **Expects** input to be sorted
  - Order log N, given N items to search through

Summary

- Introduction to search algorithms
- Linear search intuition and implementation
  - Runs in linear time
- Binary search intuition and implementation
  - Expect sorted input
  - Runs in log N time
- Next: improved sorting algorithms