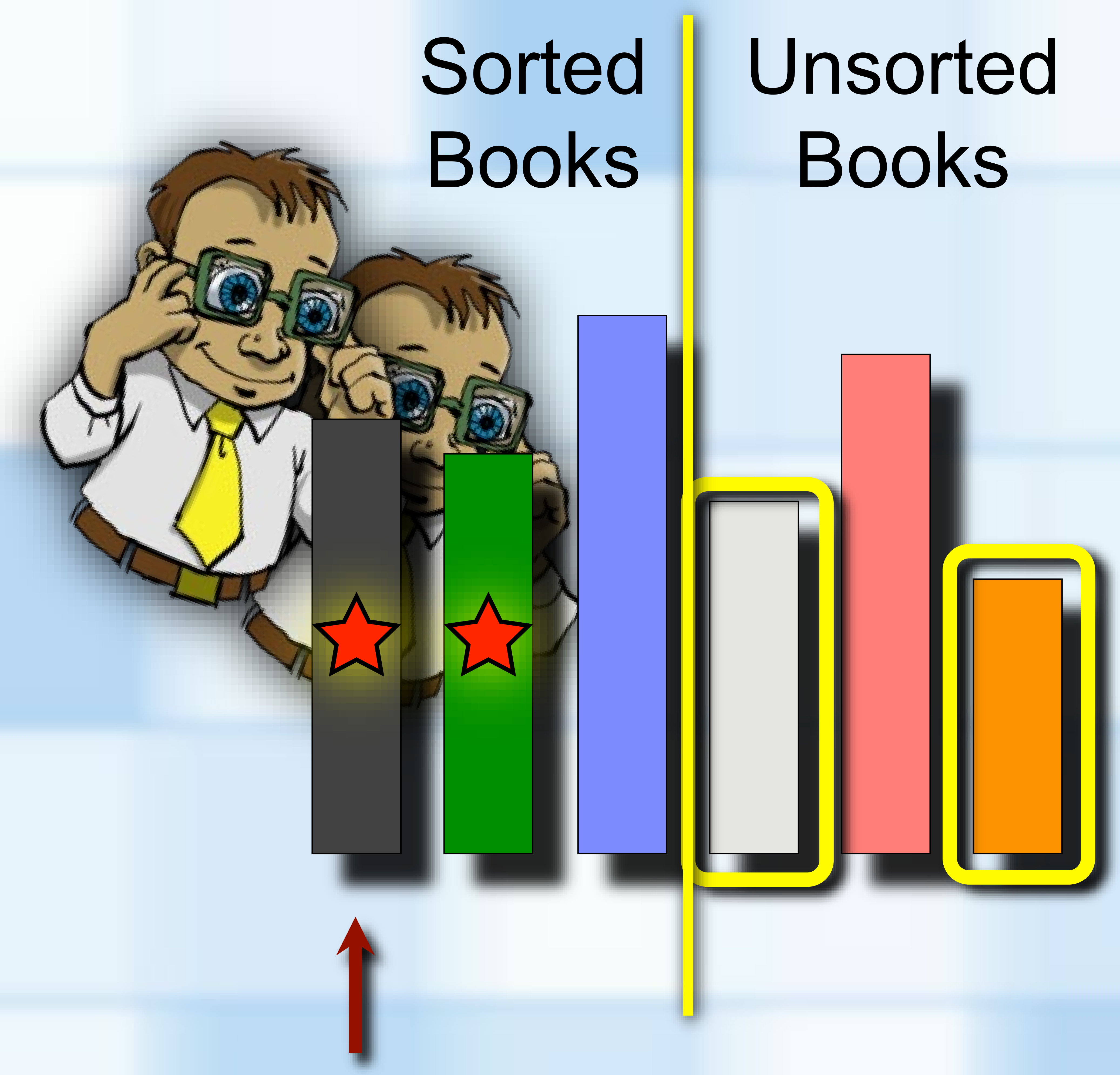
Basic Sorting Algorithms

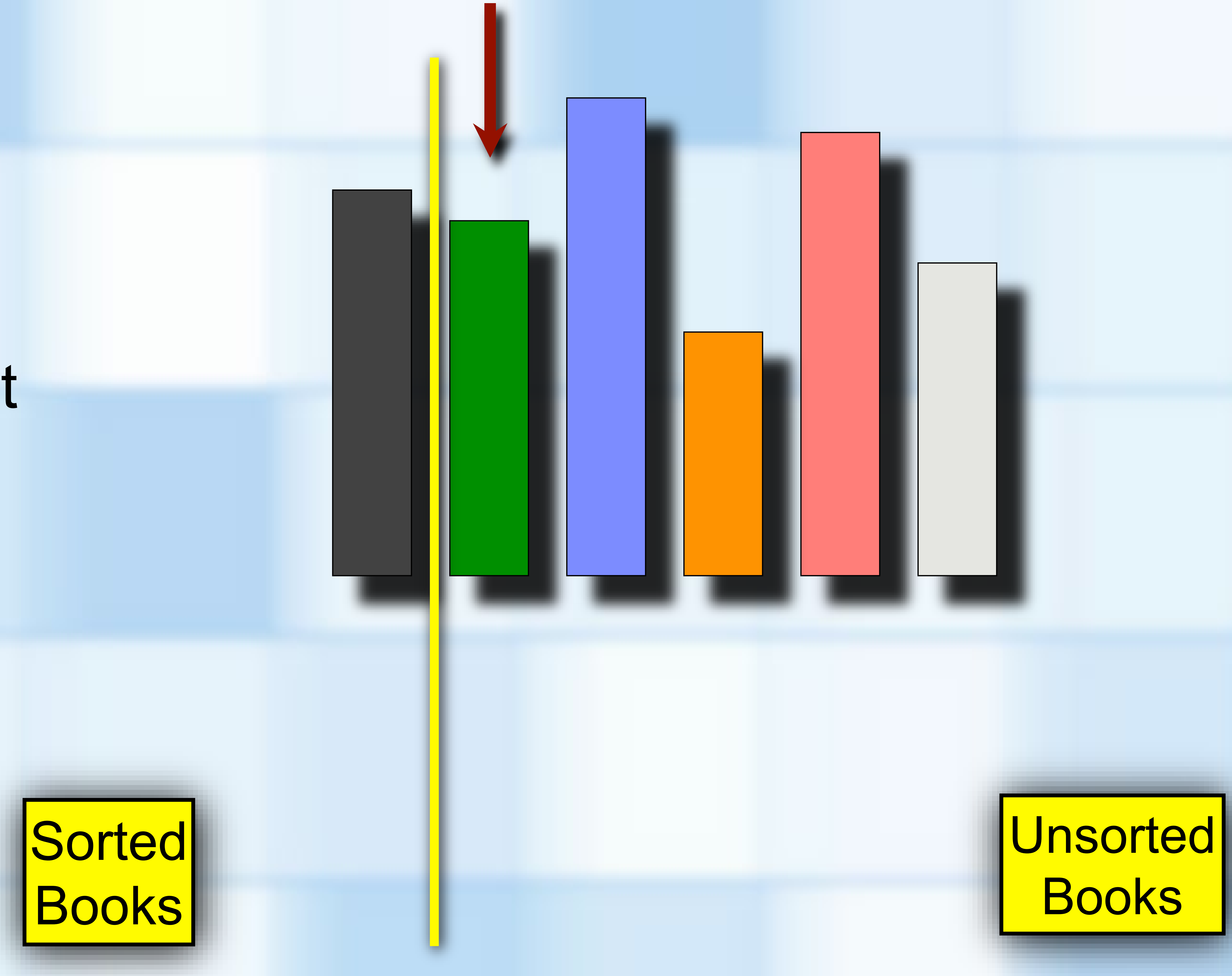
Selection Sort

- **Arrange items in order**
  - select the smallest and place it on the far left
    - starting at the first book, look at each book
    - remember the location of the smallest
    - swap it with the first book
  - repeat, starting with the second book
    - first book is in sorted portion of the shelf
  - Dividing shelf (array) into a sorted and unsorted parts
    - Partitioning



Insertion Sort

- **Insertion Sort Algorithm**
  - Leftmost item is "sorted"
  - Select the next unsorted item and remove it from the shelf
  - Move other book to the right until correct location for removed book is found
  - Insert book into it's new position.

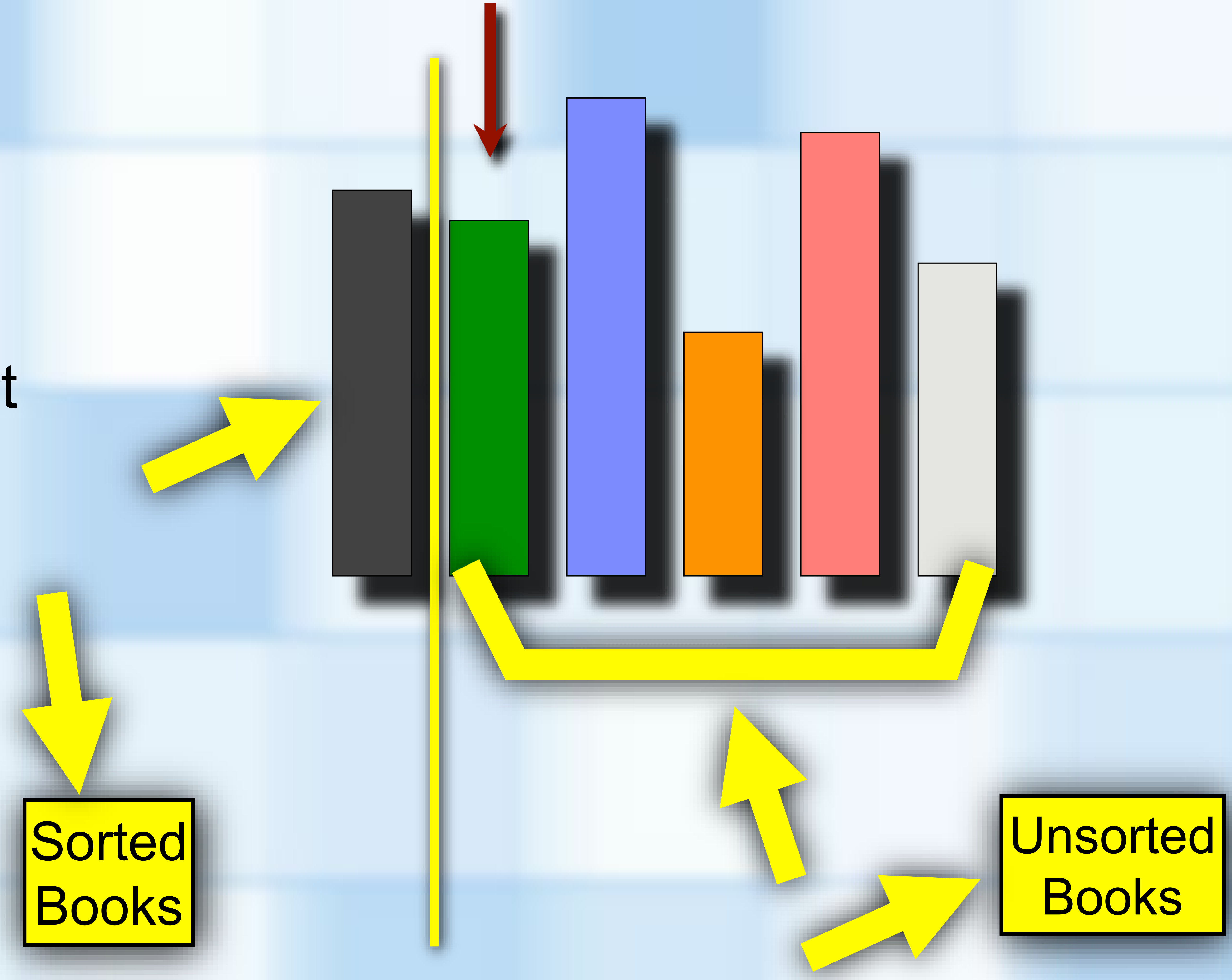


Sorted Books

Unsorted Books

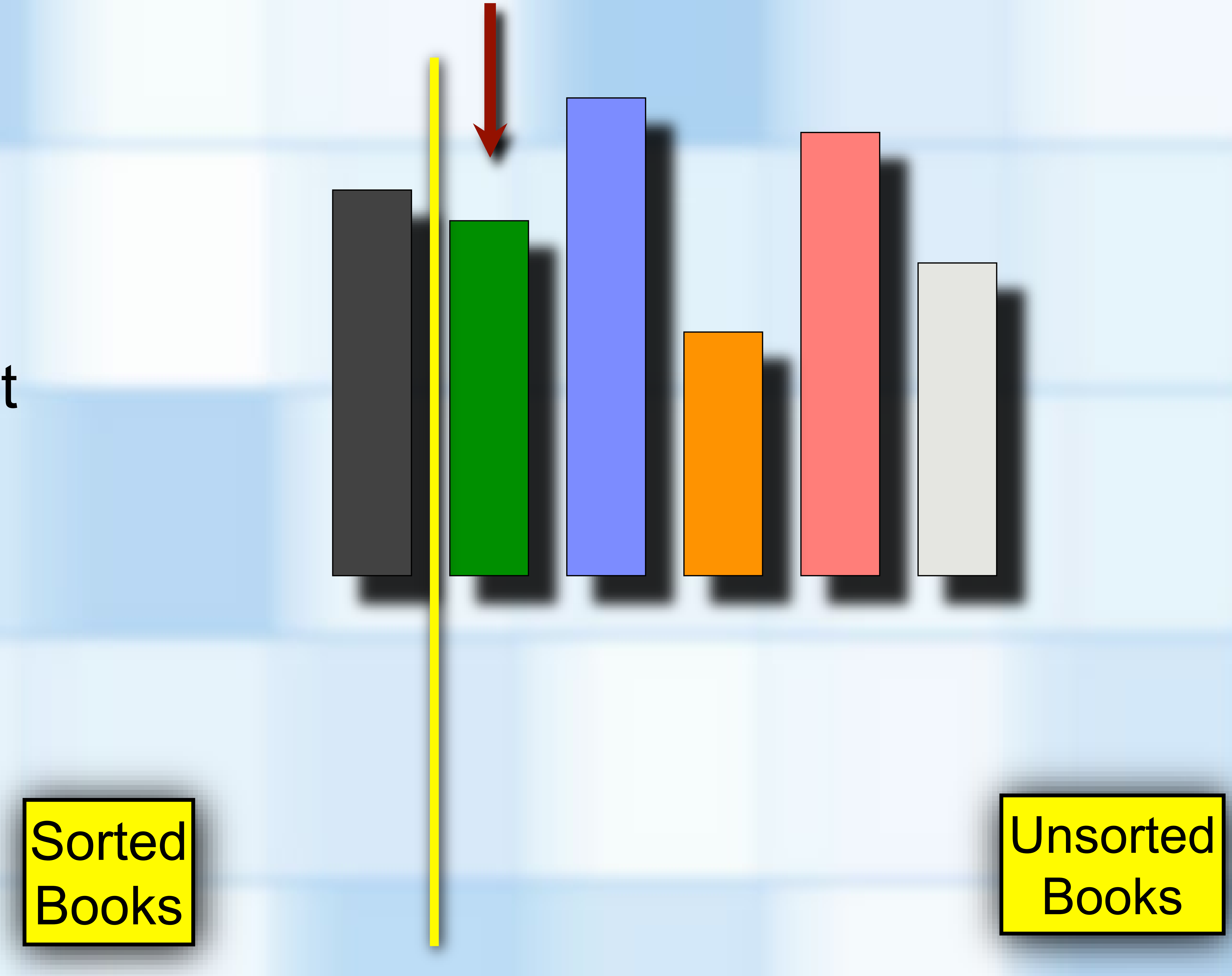
Insertion Sort

- **Insertion Sort Algorithm**
  - Leftmost item is "sorted"
  - Select the next unsorted item and remove it from the shelf
  - Move other book to the right until correct location for removed book is found
  - Insert book into it's new position.



Insertion Sort

- **Insertion Sort Algorithm**
  - Leftmost item is "sorted"
  - Select the next unsorted item and remove it from the shelf
  - Move other book to the right until correct location for removed book is found
  - Insert book into it's new position.

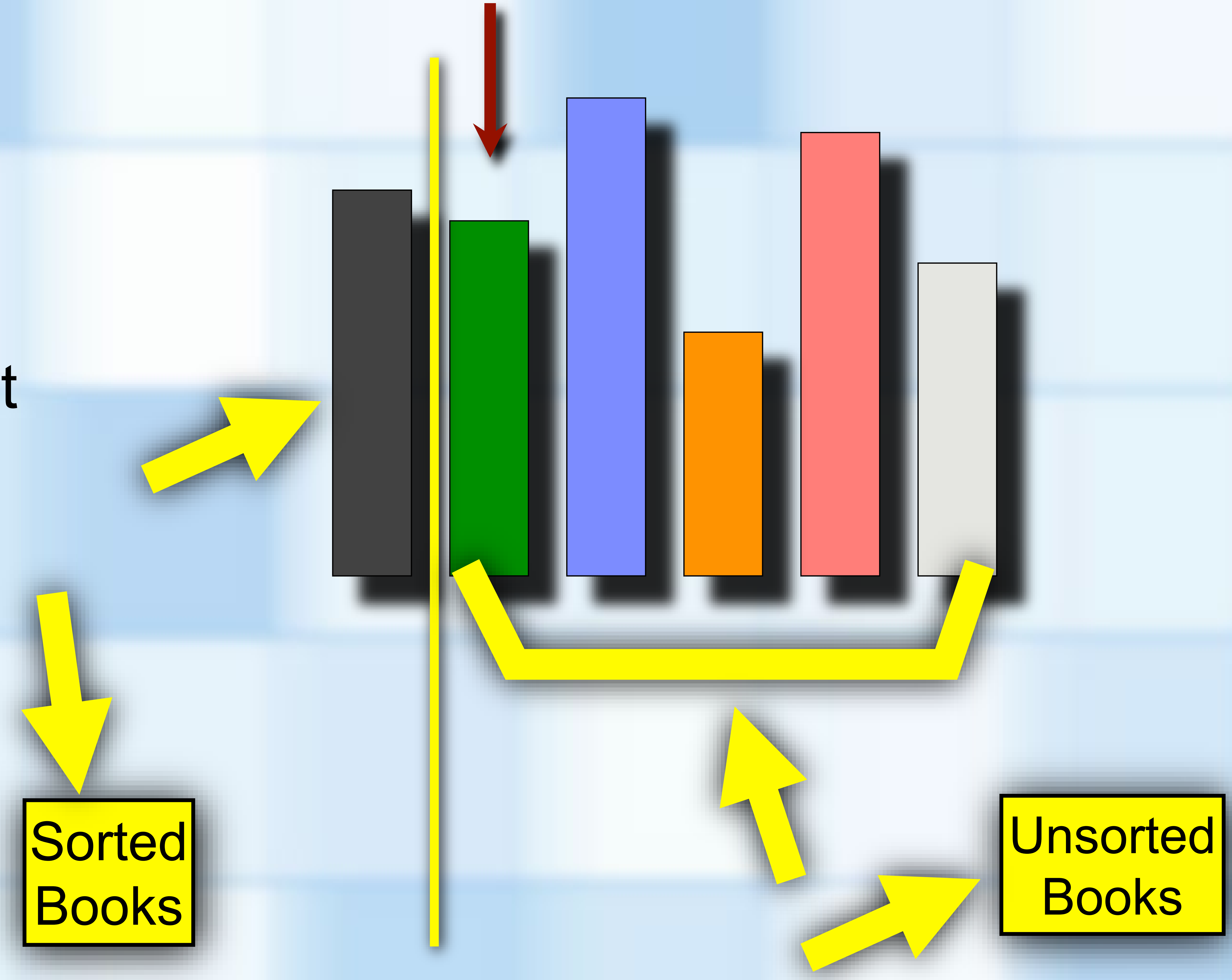


Sorted Books

Unsorted Books

INSERTION SORT

- **Insertion Sort Algorithm**
  - Leftmost item is "sorted"
  - Select the next unsorted item and remove it from the shelf
  - Move other book to the right until correct location for removed book is found
  - Insert book into it's new position.



Pearson

Insertion Sort - Arrays

- **Common Activities**
  - For each unsorted item to insert:
    - Start at the last sorted item
    - Compare it to the item to insert
    - If the item to insert is smaller,
      - move the sorted item to the right
      - compare to the next sorted item.
    - If the item to insert is larger
      (or we've reached the first element)
      - end the search and insert it

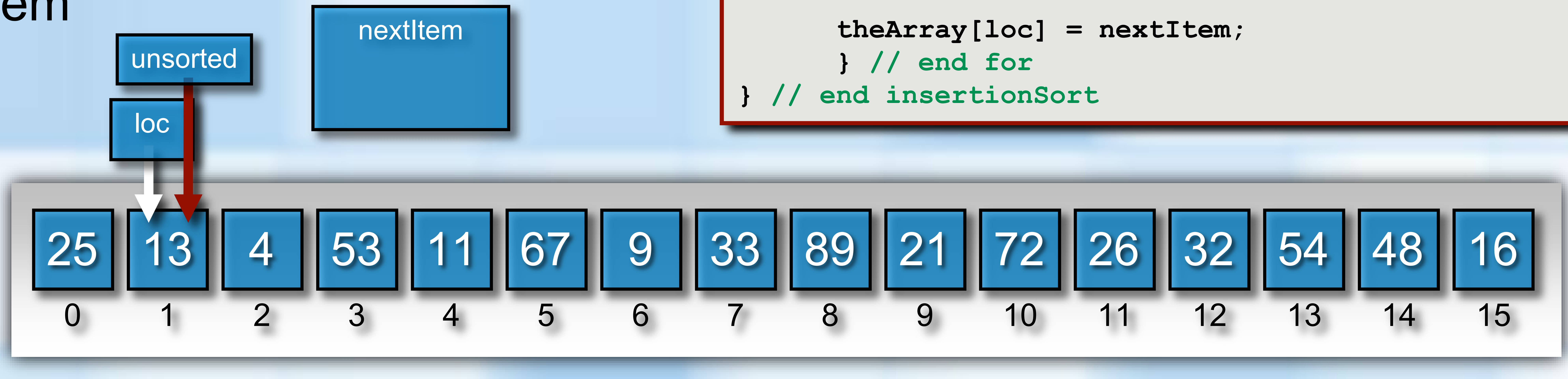
```
void insertionSort(ItemType theArray[], int n)
{
   for (int unsorted = 1; unsorted < n; unsorted++)
   {
      ItemType nextItem = theArray[unsorted];
      int loc = unsorted;
      while ((loc > 0) &&
                    (theArray[loc - 1] > nextItem) )
      {
        theArray[loc] = theArray[loc - 1];
        loc--;
      } // end while

      theArray[loc] = nextItem;
      } // end for
} // end insertionSort
```

Pearson

Insertion Sort - Arrays

- **Insertion Sort Algorithm**
  - Take the first unsorted item
  - Insert it into the sorted partition of the array
  - Repeat for each unsorted item

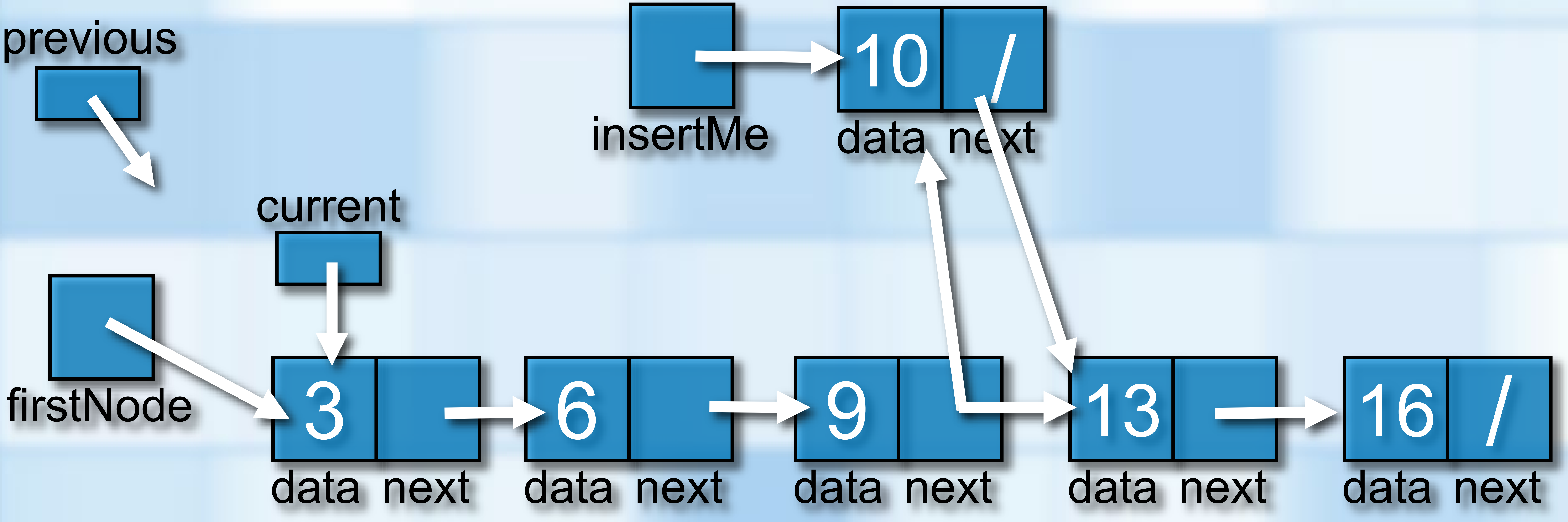
```
void insertionSort(ItemType theArray[], int n)
{
   for (int unsorted = 1; unsorted < n; unsorted++)
   {
      ItemType nextItem = theArray[unsorted];
      int loc = unsorted;
      while ((loc > 0) &&
                    (theArray[loc - 1] > nextItem) )
      {
        theArray[loc] = theArray[loc - 1];
        loc--;
      } // end while

      theArray[loc] = nextItem;
      } // end for
} // end insertionSort
```



25	13	4	53	11	67	9	33	89	21	72	26	32	54	48	16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insertion Sort - Linked Chains

- **Insertion Sort Algorithm**
  - Take the first unsorted item
  - Insert it into the sorted partition of the array
  - Repeat for each unsorted item



Insertion Sort - Linked Chains

- **Inserting into a sorted chain**
  - Initialize important variables
  - Find where to insert the node
    - Increment **`previousNode`** and **`currentNode`**
  - If **`previousNode`** is not **`nullptr`**
    - We are inserting in the middle or tail of the chain
  - If **`previousNode`** is **`nullptr`**
    - We are inserting at the head, or front, of the chain
  - Return reference to new **`firstNode`** of chain

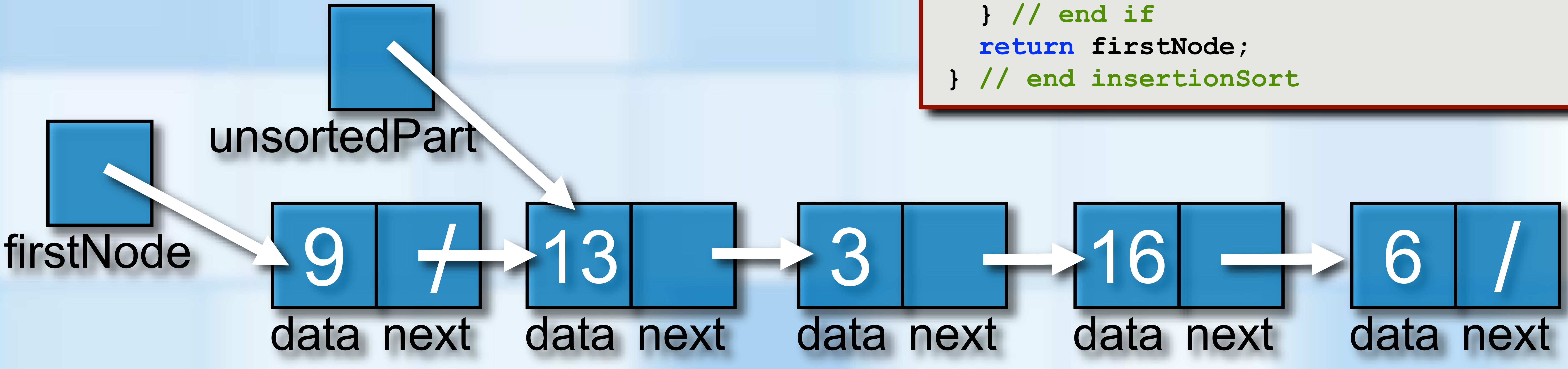
```
Node<ItemType>* insertInOrder(Node<ItemType>* firstNode,
                                                   Node<ItemType>* nodeToInsert)
{
  ItemType item = nodeToInsert->getData();
  Node<ItemType>* currentNode = firstNode;
  Node<ItemType>* previousNode = nullptr;
  // locate insertion point
  while ( (currentNode != nullptr) &&
                  (item > currentNode->getData()) )
  {
    previousNode = currentNode;
    currentNode = currentNode->getNextNode();
  } // end while
  // make the insertion
  if (previousNode != nullptr)
  {
    // insert between previousNode and currentNode
    previousNode->setNext(nodeToInsert);
    nodeToInsert->setNext(currentNode);
  }
  else // insert at beginning
  {
    nodeToInsert->setNext(firstNode);
    firstNode = nodeToInsert;
  } // end if
  return firstNode;
} // end insertInOrder
```

Pearson

INSERTION SORT - LINKED CHAINS

- **Insertion Sort Algorithm**
  - Only need to sort if there are more than two nodes
  - Break the chain into sorted and unsorted parts
  - Process each node in the unsorted chain by inserting is into the sorted chain

```
Node<ItemType>* insertionSort(Node<ItemType>* firstNode)
{
  // if zero or one item is in the chain,
  //    there is nothing to do
  if ((firstNode != nullptr) &&
                     (firstNode->getNext() != nullptr))
  {
    // break chain into 2 pieces: sorted and unsorted
    Node<ItemType>* unsortedPart = firstNode->getNextNode();
    firstNode->setNextNode(nullptr);
    while (unsortedPart != nullptr)
    {
      Node<ItemType>* nodeToInsert = unsortedPart;
      unsortedPart = unsortedPart->getNextNode();
      firstNode = insertInOrder(firstNode, nodeToInsert);
    } // end while
  } // end if
  return firstNode;
} // end insertionSort
```


unsortedPart
firstNode
9
13
3
16
6
/
data
next
data
next
data
next
data
next
data
next