CMPE 1666-Intermediate Programming

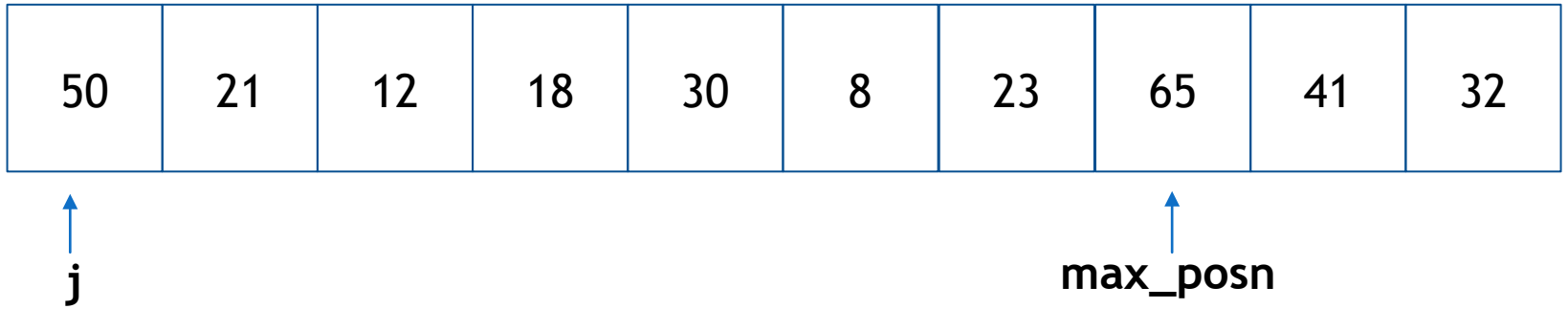
Lecture 5 – N^2- Sorting Algorithms

Selection Sort Algorithm

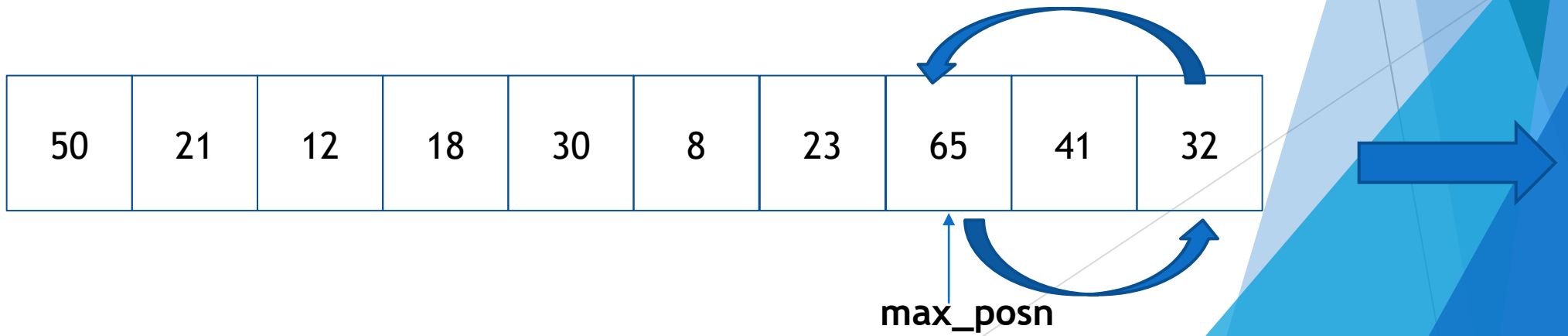
- We will consider List of n integers to be sorted in Ascending Order
- For **n elements**, the **Selection Sort** has **n-1 passes.**
- To sort in **ascending order**, in each pass we look for the **highest** value in the **unsorted** part of the List.
- At the end of the pass, this **highest value** is swapped with the element at the **highest index** in the **still unsorted part** of the List.

Selection Sort Example

- First Pass
  - We look for the index of the maximum value from index 0 to index n-1.
  - In the List below, this is the index of value **65** – index:7



  - Initially, the whole List is unsorted, so we swap with the value at position n-1.

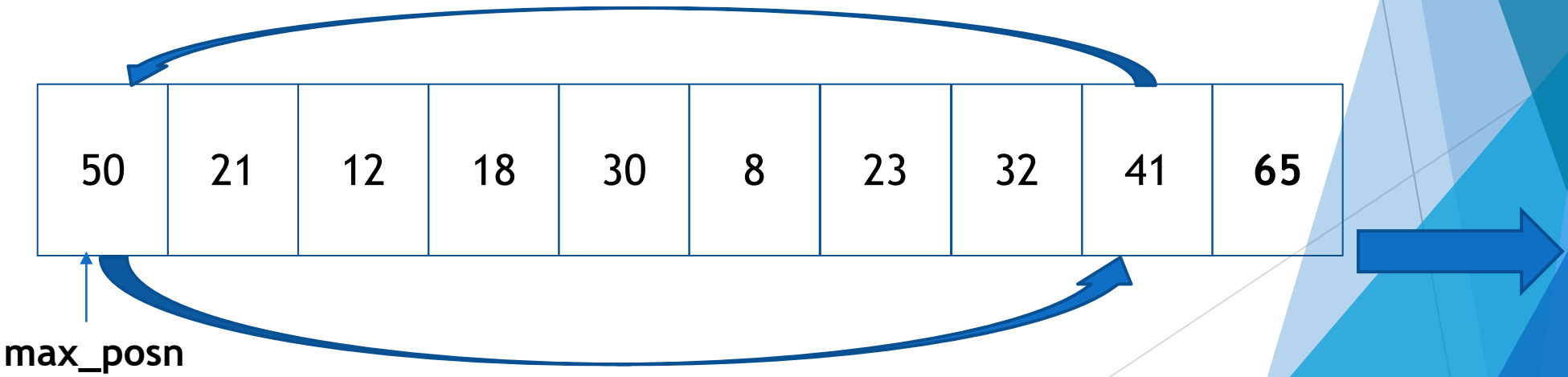


Selection Sort Example

- At the end of the first pass the highest value (in this case, **65**) is already in its position

50	21	12	18	30	8	23	32	41	**65**

- In the second pass, 50 is the maximum value in the unsorted part of the List. So it will be swapped with the second last position



Selection Sort Example

41	21	12	18	30	8	23	32	**50**	**65**

- The whole process is repeated over n-1 passes.
- To find the index of the maximum value in a pass, we start by setting **max_posn=0;**
- Then if at any position **j**, the value is higher than that at **max_posn**, we change **max_posn** to **j**.

```
if (list[j]>list[max_posn])
    max_posn=j;
```

Selection Sort Algorithm

- We are going to number our passes from 0.
- Since we have n-1 passes, our outer loop will be

  for (pass=0; pass<n-1;pass++)
- In each pass, we look for the index of the maximum value up to **n-1-pass**

```
max_posn=0;
last_posn=n-1-pass;
for (j=0; j<=last_posn;j++){
    if list[j] >list[max_posn]
        max_posn=j ;   }
```

At the end of the pass, we swap the values at the index **max_posn** with that at **last_posn**

swap(list, max_posn,last_posn)

Selection Sort Algorithm

```
for (pass=0; pass<n-1;pass++){
      max_posn=0;
      last_posn=n-1-pass;
      for (j=0; j <= last_posn;j++){
            if list[j] >list[max_posn]
                  max_posn=j ;   }

 swap(list, max_posn,last_posn)
}
```

Lecture5-Demo1

- Write a console application that contains a method that implements the Selection Sort algorithm on a list of integers.
- Write a Main method that creates a List of integer, generates 15 random integer values and adds them to the list.
- It displays the values in the order initially generated. It then uses the SelectionSort method to sort the array and displays back the sorted list.

Complexity of Selection Sort Algorithm

- The algorithm has N-1 passes.
- Pass 1 consists of N-1 comparisons
- Pass 2 consists of N-2 comparisons

……

……

……

- Pass N-1 consists of 1 comparison
- The total no. of comparisons is: (N-1) + (N-2) +……+ 1=1/2 (N)(N-1)
- $=1/2(N^2-N)$
- The complexity of the algorithm is thus $O(N^2)$

Insertion Sort

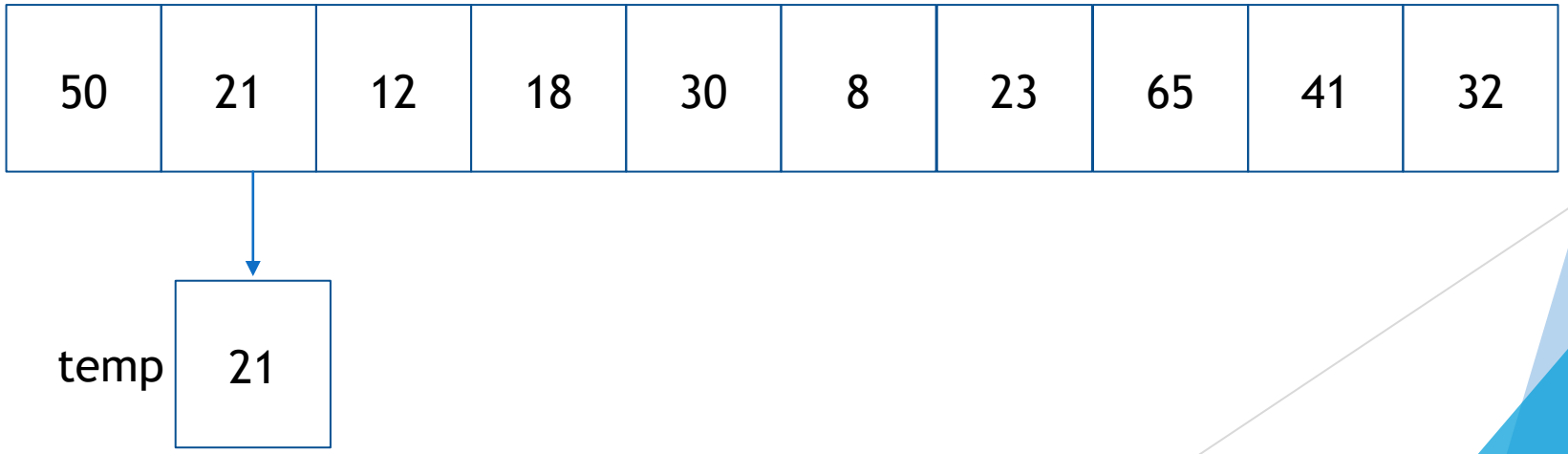
- We are again going to consider a list of n integer values, to be sorted in ascending order.
- In the Insertion sort, we start sorting from the Left-hand side of the list.
- When sorting in ascending order, we move each element **as far to the left as possible** as long as there are elements bigger than it on this left.
- Again we are going to have **n-1** passes

Insertion Sort Example

- Consider the list below

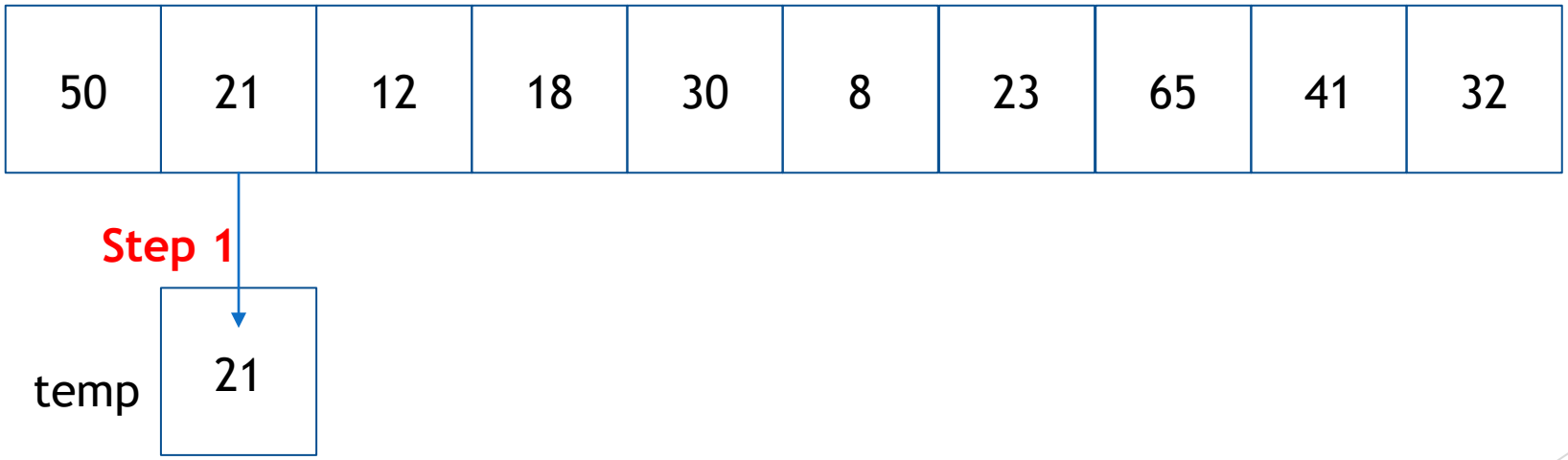
50	21	12	18	30	8	23	65	41	32

- In the first pass, we start with the second element (21) and copy it to a temp variable. We'll then compare it with the value behind it. If that value is bigger than the temp variable, we move the value one position to the right.

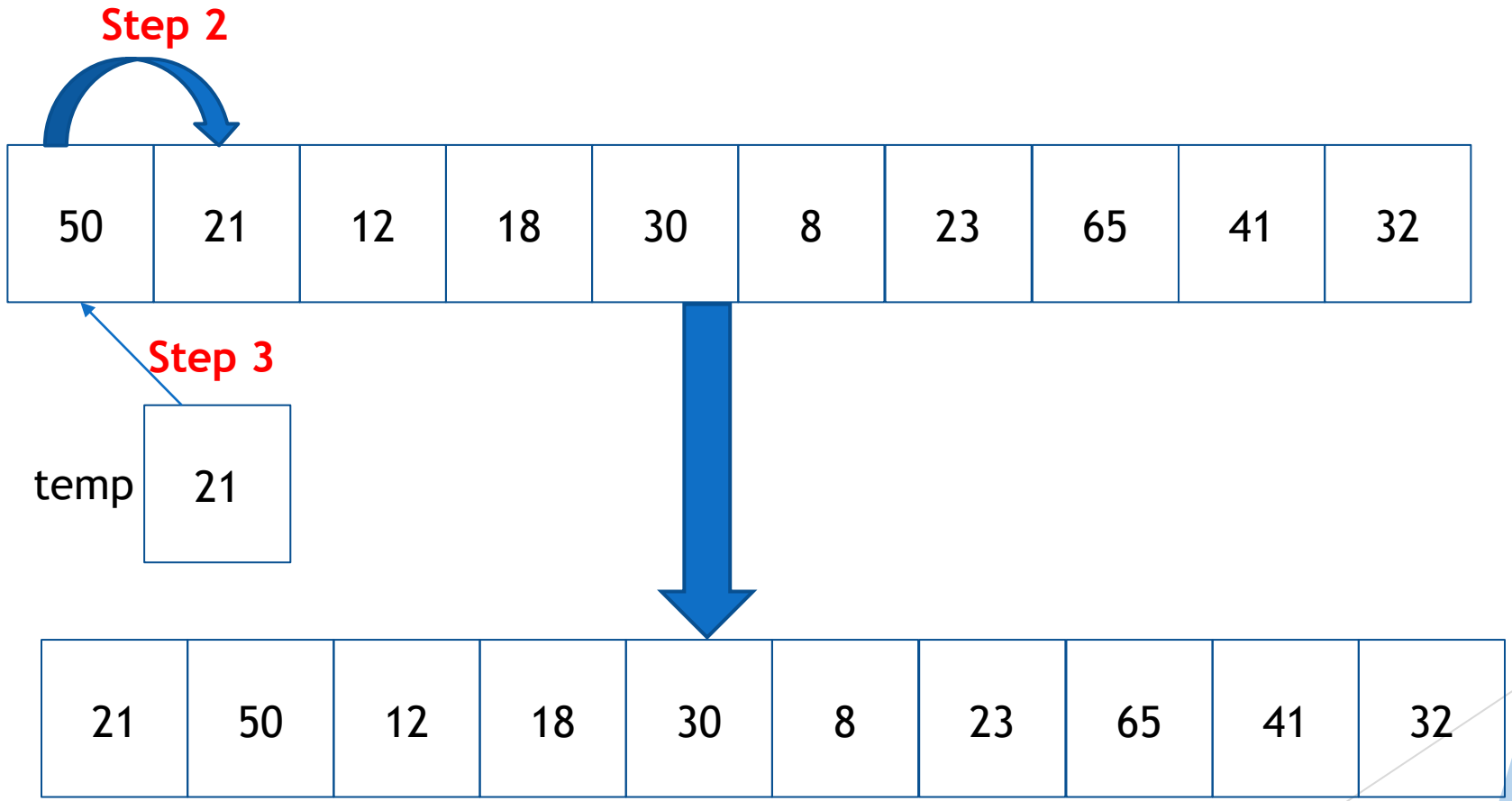


Insertion Sort Example

- we copy 21 into a temporary variable
  - Since 50 > 21, Move 50 one position to the right
  - Then place 21 in the position previously occupied by 50



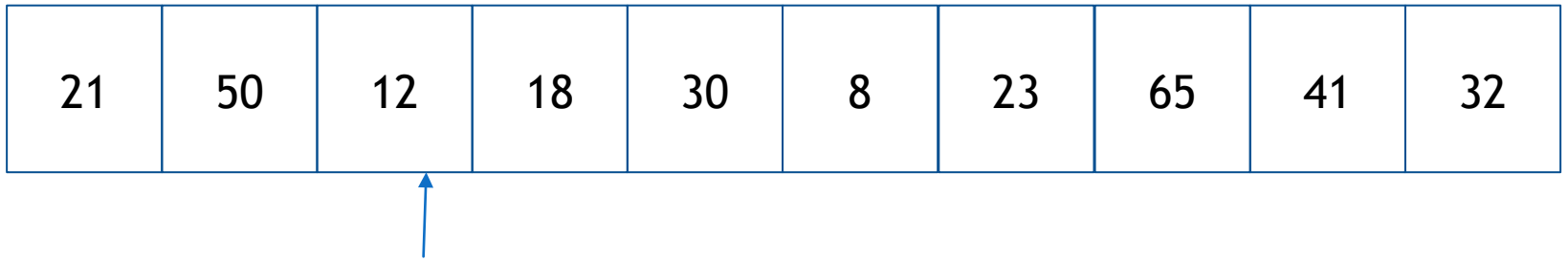
Insertion Sort Example

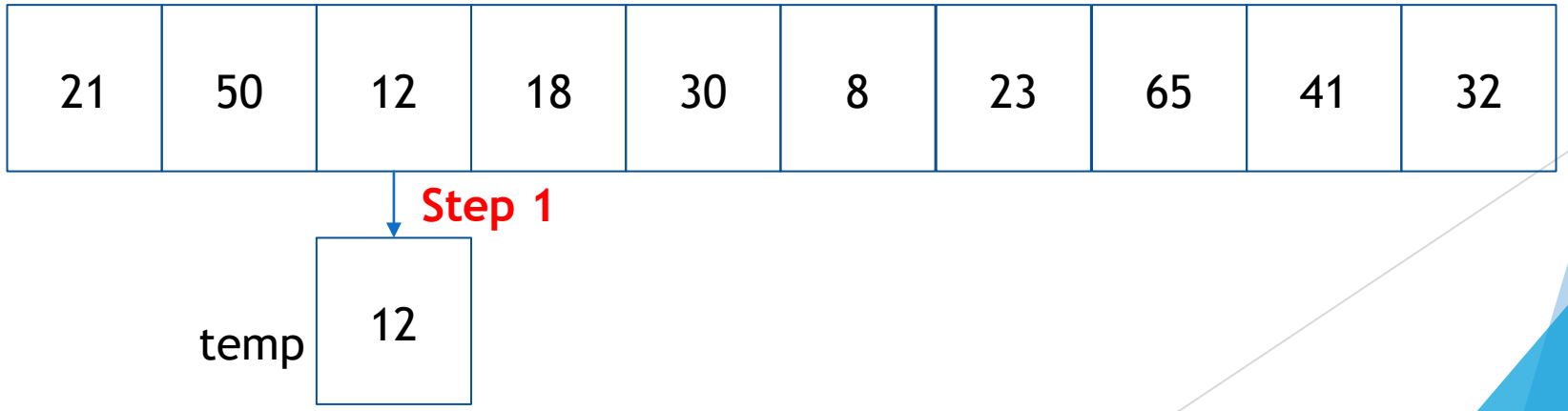


- The first pass is over

Insertion Sort Example

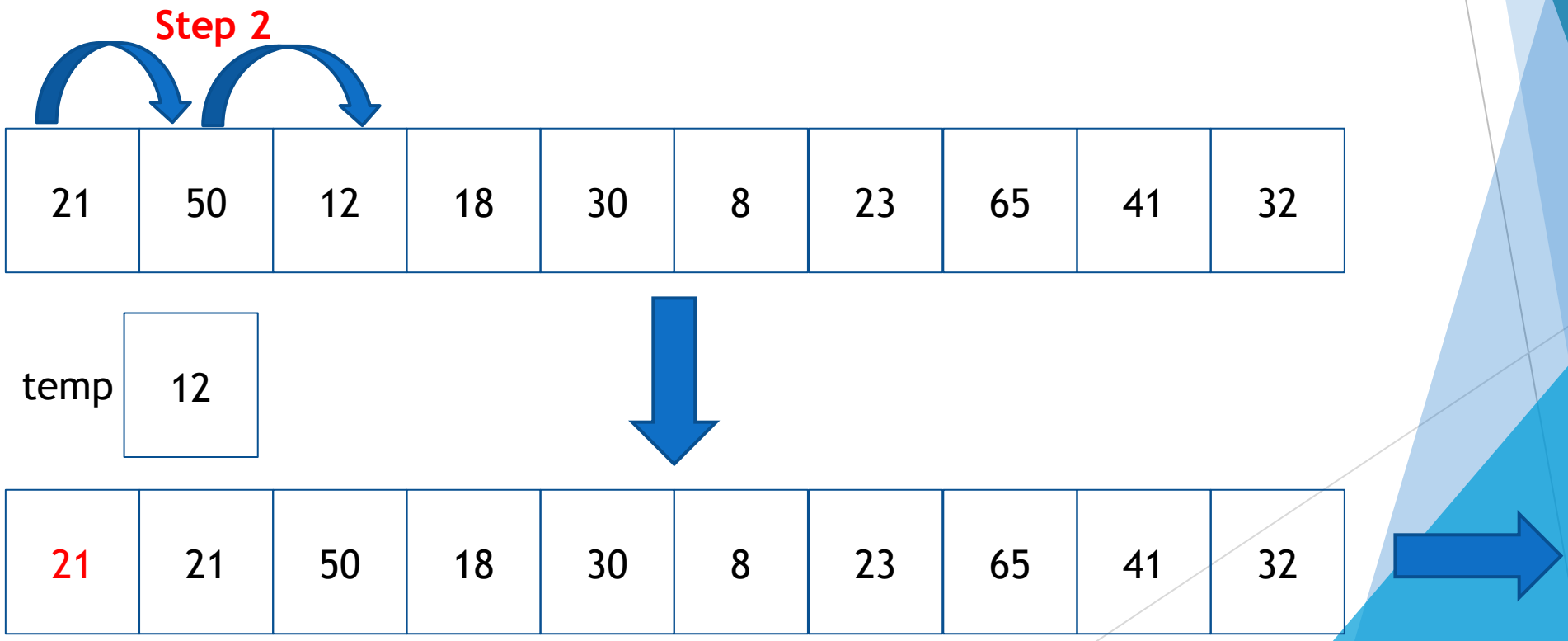
- In the second pass, we pick the next element (12).
- we move 12 into the temporary variable



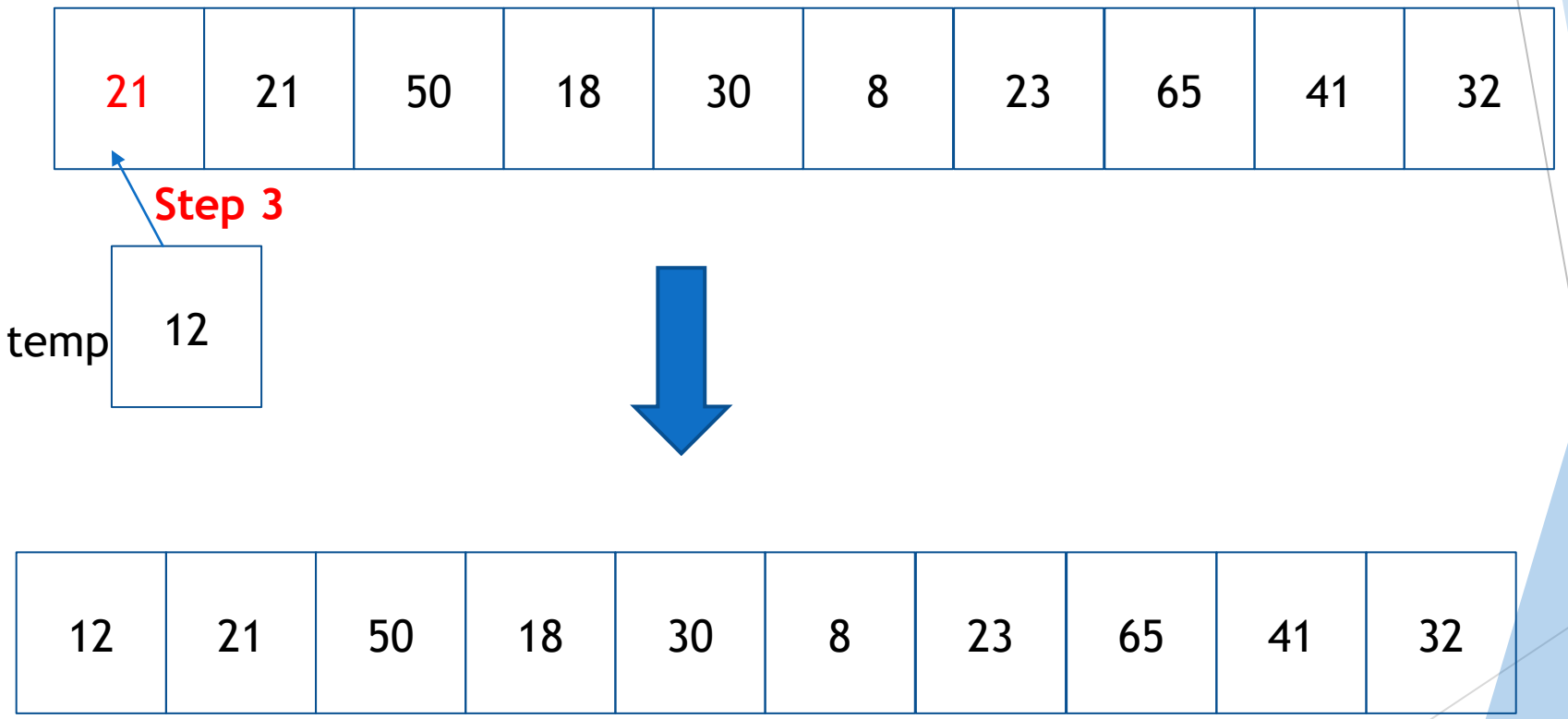


Insertion Sort Example

- We next compare the value of temp with all values that were before it in the List, as long as the elements are bigger.
- All the bigger elements are shifted one position to the right

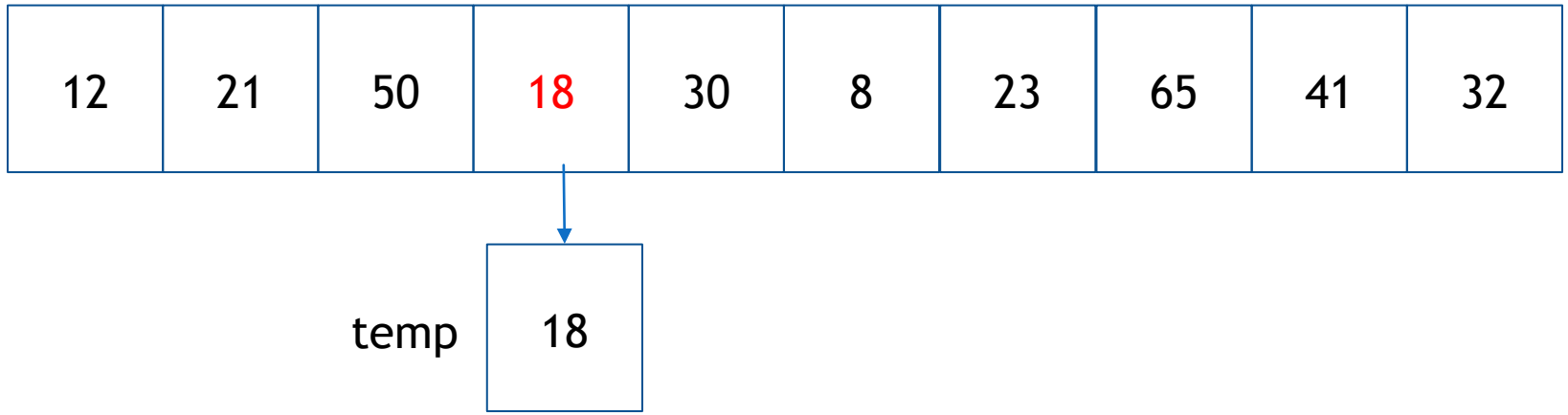


Insertion Sort Example

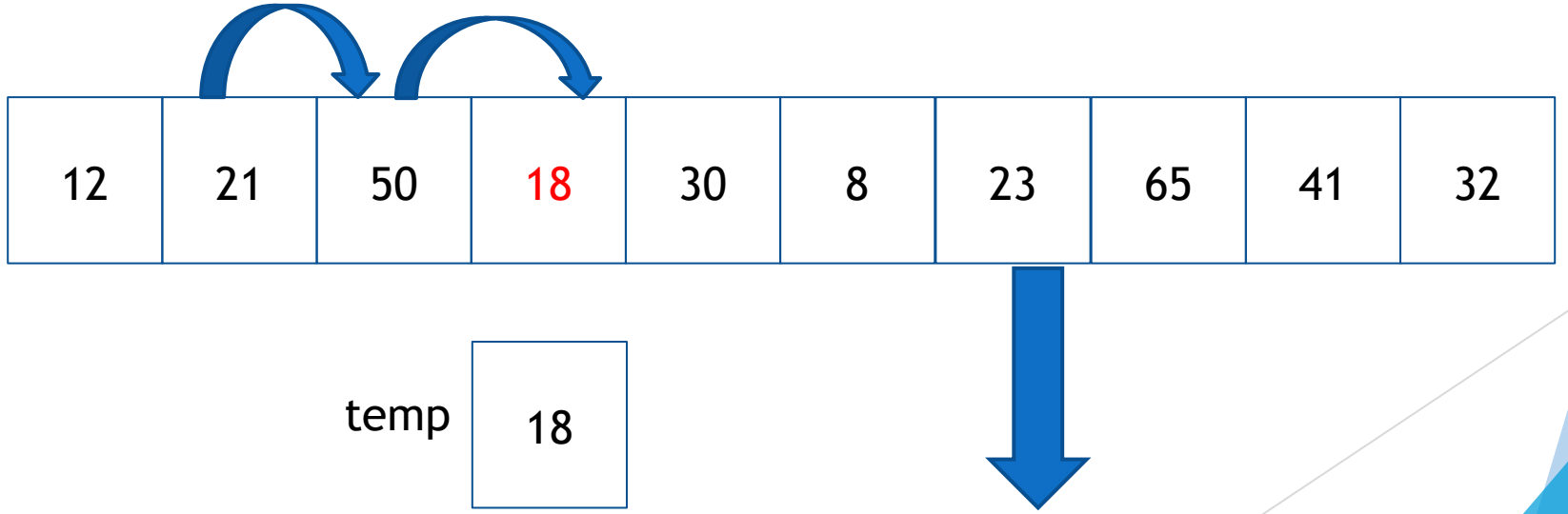


Insertion Sort Example

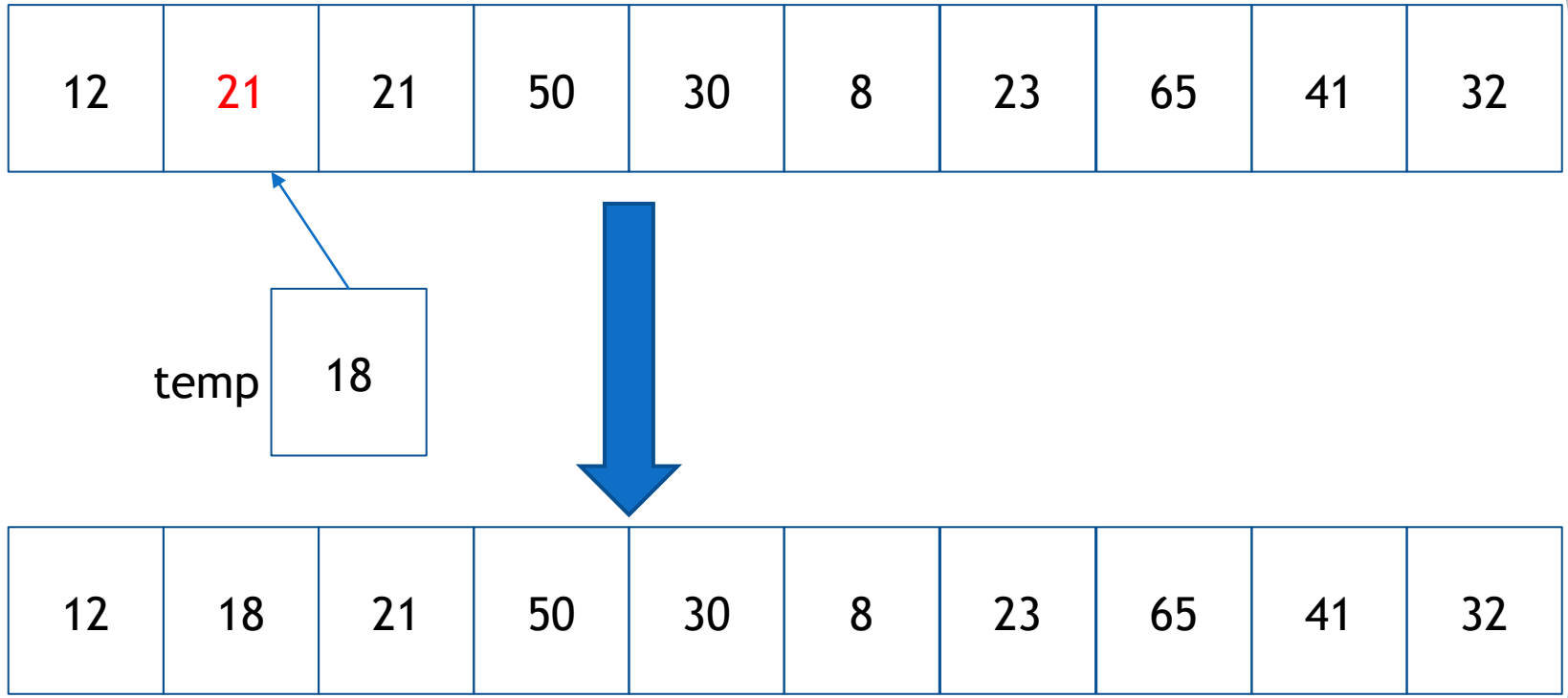
- In the next pass, we take value 18, again we compare with the previous value



- Since 50>18, we repeat the steps in a similar way to what we had previously done



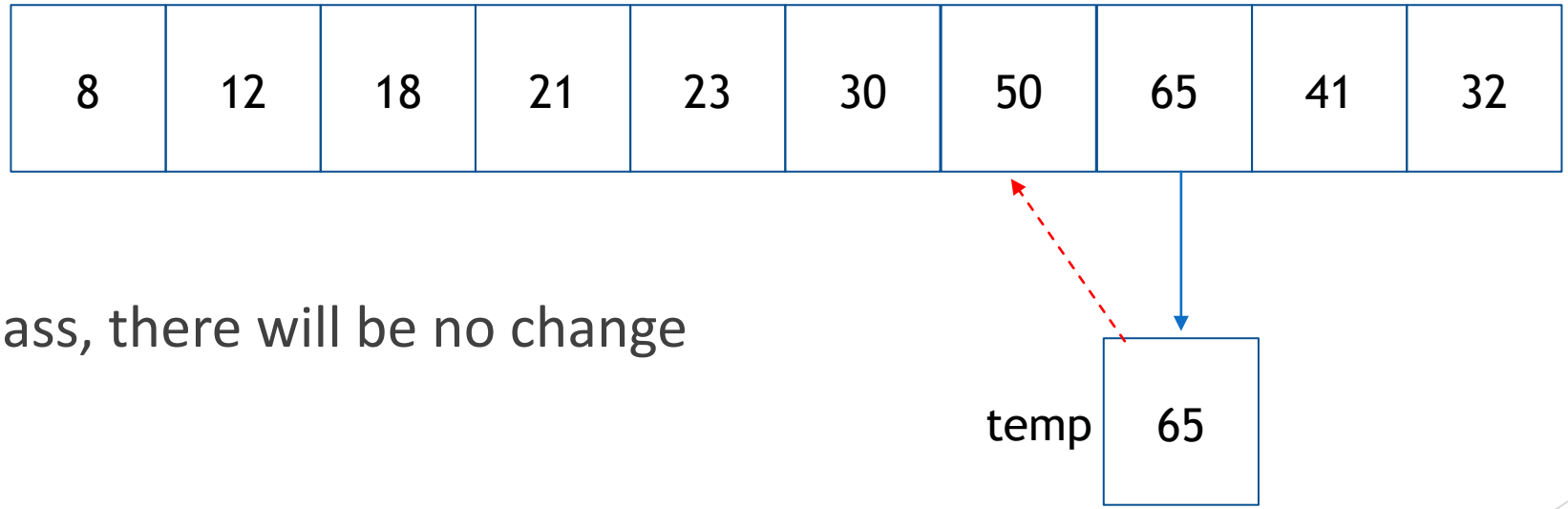
Insertion Sort Example



- Note that in this case since 12 <18, 12 is not shifted.

Insertion Sort Example

- Note that at some point, we'll have the situation below, where the next value considered is 65.
- When we compare with the previous value, 50, it is less than 65.



- In this pass, there will be no change

Insertion Sort Algorithm

- Thus for a list of n elements, we have n-1 passes.
- In each pass, **p** (p=1..n-1), we compare **list[p]** with **list[j]** where **j<p**, and **list[j]>list[p]**
- For each such **list[j]**, we move **list[j]** one position to the right

  list[j+1]=list[j]
- Hence the algorithm is

```
for (p=1;p<n;p++)
    temp=list[p]
    j=p-1;
   while ((j>=0) && (list[j] >temp))
            list[j+1]=list[j];
             j=j-1;
    endwhile
    list[j+1]=temp;
endfor
```

Complexity of the Insertion Sort Algorithm

- For n elements, the algorithm has n-1 passes.
- In the 1st pass there is one comparison and possibly 2 elements moved
- In the 2nd pass, there is potentially 2 comparisons and 3 elements moved
- In the (n-1)th pass, there is potentially n-1 comparisons and n elements moved
- So, the total number of comparisons is: 1+2+3……+ n-1= n(n-1)/2 giving us an $O(n^2)$ algorithm

Lecture5-Demo1B

- Add a method to Demo1 to sort a list of integers on the Insertion Sort technique.
- Add the required code to Main() to create a second list of 15 random integers and to sort it using the InsertionSort method.
- Again display the initial list and the final list.

Comparing Strings

- *string.Compare(myString1, myString2)*
- **Return type:** The return type of this method is Integer. It will be:
  - Less than zero: If the first string is lexicographically smaller than the second string.
  - zero: If both strings are equal.
  - Greater than zero: If the first string is lexicographically greater than the second string.

Comparing Strings

- Using CompareTo() method
- *myString1.CompareTo(myString2)*
- **Return type:** The return type of this method is Integer. It will be:
  - Less than zero: If the first string is lexicographically smaller than the second string.
  - zero: If both strings are equal.
  - Greater than zero: If the first string is lexicographically greater than the second string.

Using Insertion Sort on a list of strings

```
Procedure InsertionSort(list) //list is a list of strings
Begin
    n=sizeOf(list);
     for (p=1;p<n;p++)
          temp=list[p]
          j=p-1;
           while ((j>=0) && (list[j].CompareTo(temp)>0))
               list[j+1]=list[j];
               j=j-1;
           endwhile
          list[j+1]=temp;
    endfor
End #of procedure
```

Lecture5- Demo1C

- To Demo1, add a method to sort a List of strings using the Insertion Sort technique
- In **Main()**, declare and create a List of 15 strings.
- Add the required code to Main to call the **InsertionSort()** method to sort the List of strings
- Perform the display of the List before and after the sorting.

Sorting a list of Structs

- We have seen that when sorting a list (or array) of integers, we compare the values in different elements of the list and swap where required.
- When we sort a list or an array of a struct type, we have to choose a member on which we perform the comparison. Then we perform the sorting on values of that member.
- For example to sort a list of Student structs, we can compare the student Id values then sort the array or list of structs based on the ID values.
- The next slide gives the algorithm for implementing SelectionSort on a list of Student struct, using the studentdId value. We will assume the following struct:

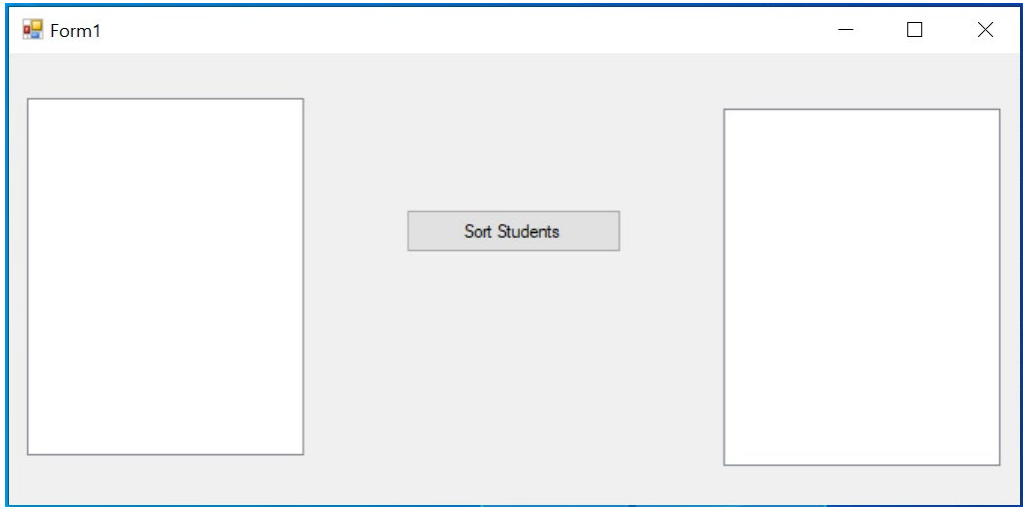
```
struct Student{
    public int studentId;
    public string firstName;
    public string lastName;
   }
```

Using Selection Sort to sort a list of Student Structs on StudentId

```
Procedure SelectionSort(list) //list is a list of strings
begin
    for (pass=0; pass<n-1;pass++)
        max_posn=0;
        last_posn=n-1-pass;
        for (j=0; j <= last_posn;j++){
            if (list[j].studentId >list[max_posn].studentId)
                max_posn=j ;
            endif
        endfor
        Swap(list,max_posn,last_posn)
     endfor
end
```

Lecture5-Demo2

- You have been provided with a starter program with the UI shown below.



- The program already contains the definition of a struct Student and the declaration and initialization of a list of such structs, a **CreateStudent()** method that creates and returns a struct and a **StudentToString()** method that returns the information in a Student struct as a string.
- It also contains 3 arrays of information, named **stdIdArr**, **firstNames**, and **lastNames**, that respectively contain the student Id's, first name and last name of a number of students.
- Add the code as specified in the next slide.

Lecture5-Demo2-contd

- To the Form class add, as member variable, a List of Student structs. Include the code to create an empty List.
- Add the code such that on Form load, intances of Student structs are constructed from the 3 arrays, using the **CreateStudent()** method and added to the List. The student structs must also be displayed in the left listbox using the **StudentToString()** method.
- Add an adaptation of the **Selection Sort** method to sort a list of **Student** structs on the **studentId** data member.
- Add a click event handler for the button, such that it makes a copy of the Student List, sorts the copy using the implemented **SelectionSort()** method and displays the sorted list in the right listbox.