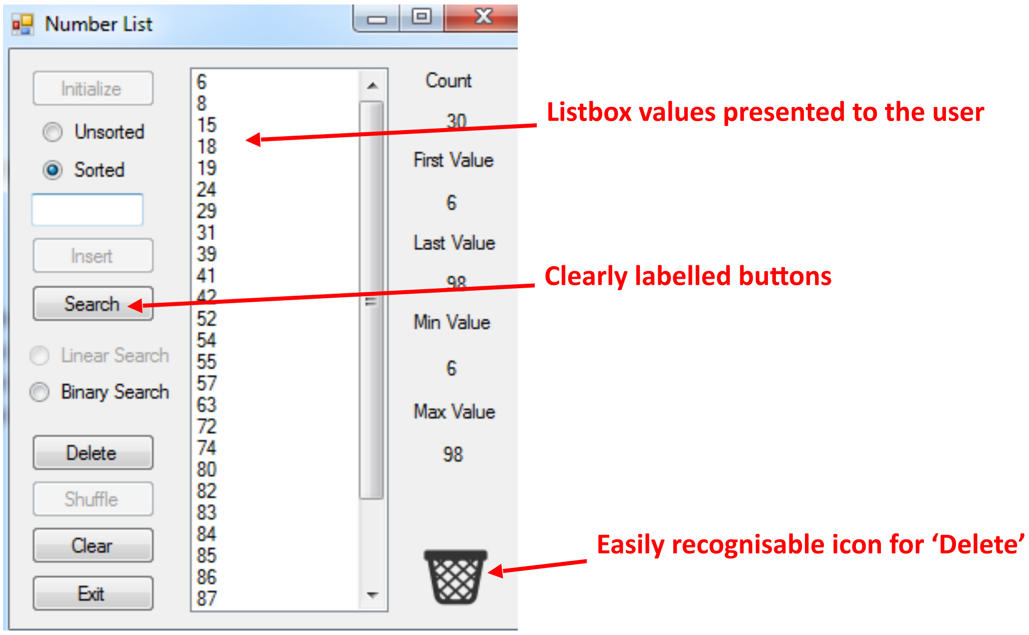
Event Driven & Graphical User Interface Programming

Number List Manager

Design

Presentation & Accessibility

To create my Number List program following the specified requirements, I first decided that the windows form should look presentable and display what each bit of functionality does e.g. clearly labelled buttons. The form should also display the contents of the listbox to the user at runtime making it easier to access the elements of the listbox using functionality such as insert, search, delete etc.



○ Unsorted
● Sorted

○ Linear Search
○ Binary Search

6
8
15
18
19
24
29
31
39
41
42
52
54
55
57
63
72
74
80
82
83
84
85
86
87

Count 30

First Value 6

Last Value 98

Min Value 6

Max Value 98

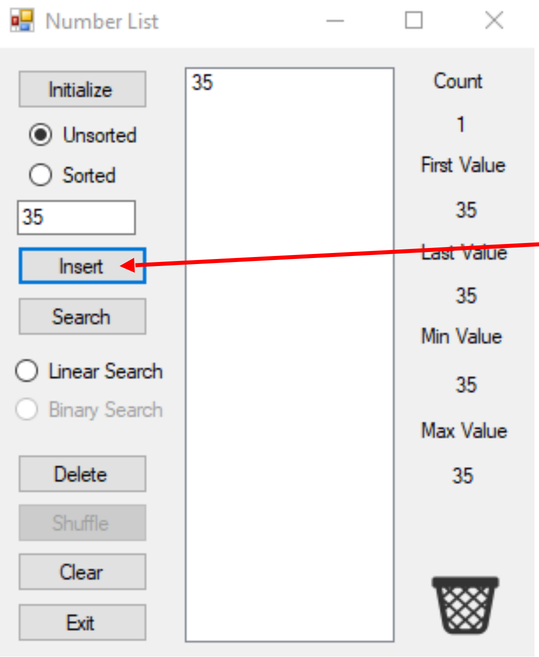
Listbox values presented to the user

Clearly labelled buttons

Easily recognisable icon for 'Delete'

I also decided that the program shall dynamically disable/enable buttons, radio buttons and functionality at runtime. To achieve this programmatically, I created a method that checked for conditions such as disabling 'Insert' when the listbox reached the max count of 30 for example.

This was beneficial as it stopped users from wasting time trying to complete actions that the program was not intended to do. Also, this helped me from implementing multiple error message prompts into the code design which prevented the program from being overwhelming with message prompts for both the user and the designer.



● Unsorted
○ Sorted

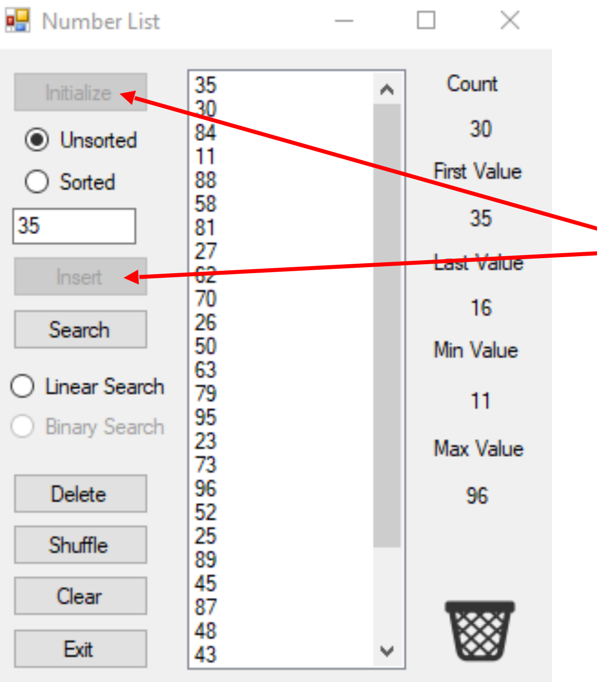
35

○ Linear Search
○ Binary Search

35

Count
1
First Value
35
Last Value
35
Min Value
35
Max Value
35

Insert working as intended as there's less than 30 elements contained in the listbox



● Unsorted
○ Sorted

35

○ Linear Search
○ Binary Search

```
35
30
84
11
88
58
81
27
62
70
26
50
63
79
95
23
73
96
52
25
89
45
87
48
43
```

Count
30
First Value
35
Last Value
16
Min Value
11
Max Value
96

Initialize and Insert both disabled as there are 30 elements contained in the listbox

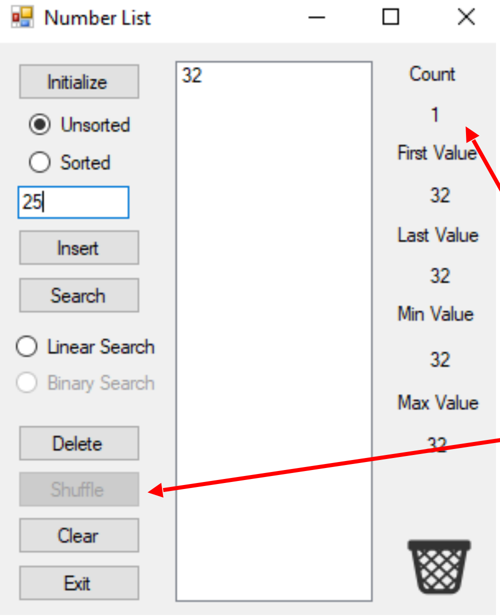
For the errors I couldn't prevent such as incorrect input into a textbox or attempting to insert a value outside the range of 0-100, I used MessageBox prompts to display error messages to the user. I also ensured that the error messages clearly described the issue e.g. "Please enter a valid number within the range 0-100!" instead of an error message saying "Error!" as this gives no useful information to the user.

Implementation

There are two methods 'btnCondition' and 'listStatus' that appear regularly in other methods, so it is best that these are shown first.

btnCondition

'btnCondition' allows the program to dynamically disable and enable controls such as buttons and radio buttons at runtime. This is beneficial as it stops the user from trying to click controls that shouldn't be enabled or used at that specific moment e.g. attempting to shuffle an empty list. This also allows the use of less message prompts and error checking in specific controls as they will not be executed when disabled. As shown below, the program will not allow the 'Shuffle' control to be enabled if the listbox size is less than 2. In this case, the listbox only contains 1 value so it is not possible to shuffle the items within the list.



● Unsorted

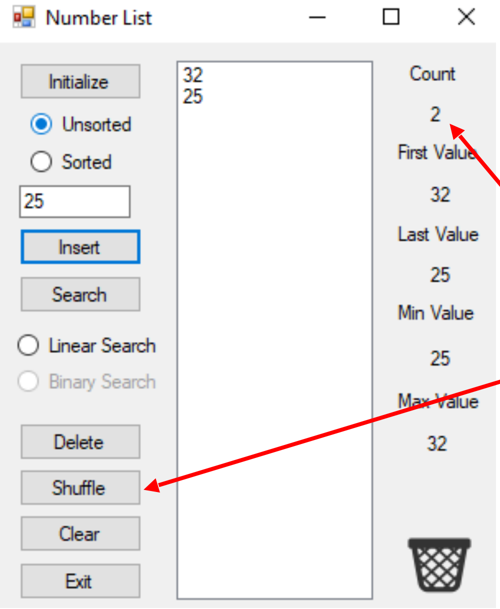
25

32

Count
1
First Value
32
Last Value
32
Min Value
32
Max Value
32

Count is less than 2 therefore 'Shuffle' is disabled

If there are 2 or more items within the list, it will enable the 'Shuffle' control as it's possible to shuffle 2 items.



● Unsorted

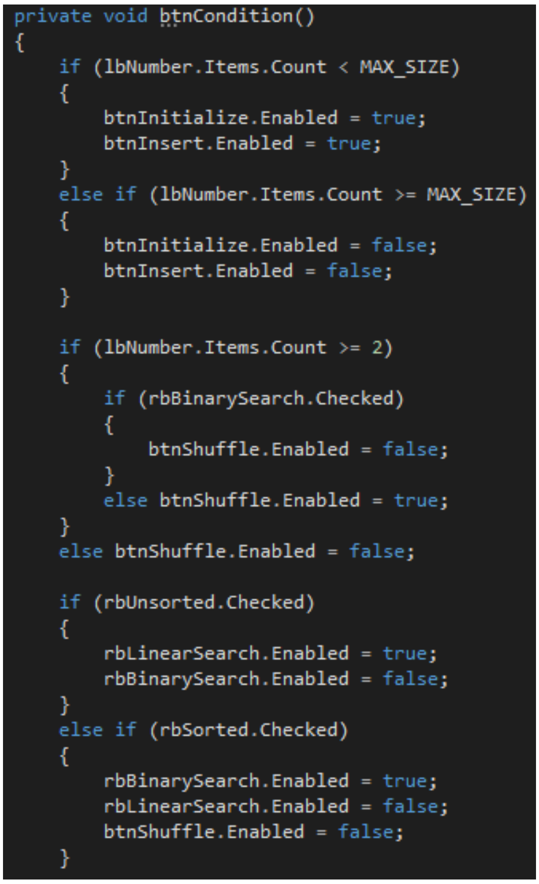
25

32
25

Count
2
First Value
32
Last Value
25
Min Value
25
Max Value
32

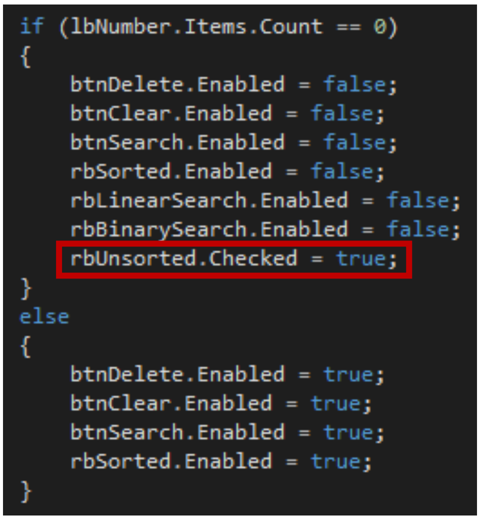
Count is equal to 2 therefore 'Shuffle' is enabled

‘btnCondition’ used if else statements to decide whether certain controls should be enabled or disabled based on certain conditions such as the size of items contained within the list or whether a specific option was selected such as ‘rbSorted’ which disabled the use of ‘btnShuffle’ as the list ensures it stays sorted when ‘rbSorted’ is enabled.



```
private void btnCondition()
{
    if (lbNumber.Items.Count < MAX_SIZE)
    {
        btnInitialize.Enabled = true;
        btnInsert.Enabled = true;
    }
    else if (lbNumber.Items.Count >= MAX_SIZE)
    {
        btnInitialize.Enabled = false;
        btnInsert.Enabled = false;
    }

    if (lbNumber.Items.Count >= 2)
    {
        if (rbBinarySearch.Checked)
        {
            btnShuffle.Enabled = false;
        }
        else btnShuffle.Enabled = true;
    }
    else btnShuffle.Enabled = false;

    if (rbUnsorted.Checked)
    {
        rbLinearSearch.Enabled = true;
        rbBinarySearch.Enabled = false;
    }
    else if (rbSorted.Checked)
    {
        rbBinarySearch.Enabled = true;
        rbLinearSearch.Enabled = false;
        btnShuffle.Enabled = false;
    }
```

As controls are enabled/disabled at runtime programmatically, there had to be conditions that ensured that certain controls shouldn’t be enabled when the program first launches (or technically when the list is equal to 0).



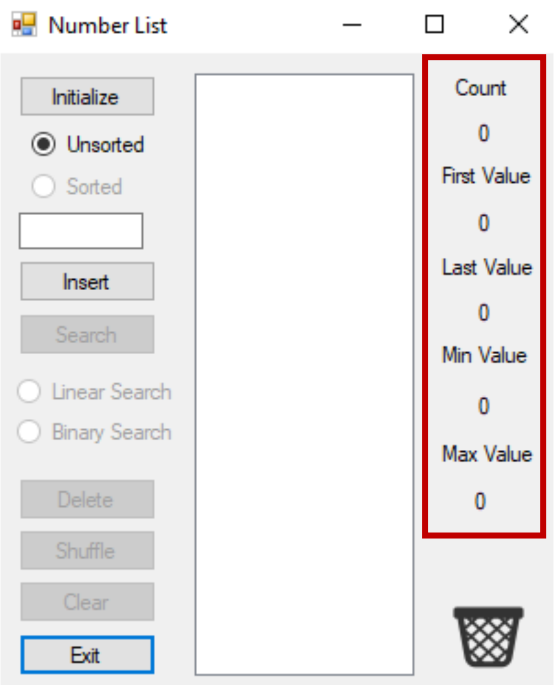
```
if (lbNumber.Items.Count == 0)
{
    btnDelete.Enabled = false;
    btnClear.Enabled = false;
    btnSearch.Enabled = false;
    rbSorted.Enabled = false;
    rbLinearSearch.Enabled = false;
    rbBinarySearch.Enabled = false;
    rbUnsorted.Checked = true;
}
else
{
    btnDelete.Enabled = true;
    btnClear.Enabled = true;
    btnSearch.Enabled = true;
    rbSorted.Enabled = true;
}
```

As highlighted, if the list is empty, rbUnsorted will be force checked as this prevents any bugs such as 'Sorted' still being checked when the button itself is disabled. This also allows me to use less error checking within the code as certain methods will not be allowed to execute.

listStatus

'listStatus' allowed the use of statistics to be shown at runtime showing stats within the list ranging from smallest and highest value, first and last value and the total size of items contained in the list.

Before

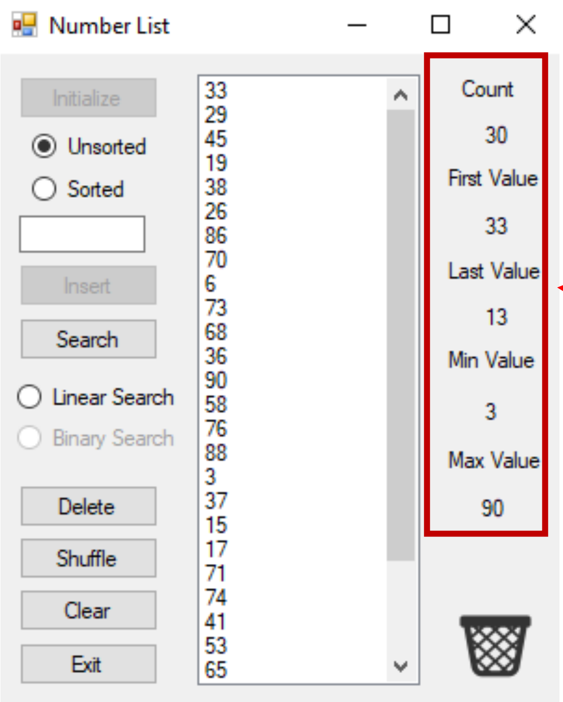


● Unsorted
○ Sorted

Count
0
First Value
0
Last Value
0
Min Value
0
Max Value
0

List is empty; therefore, all stats are equal to 0

After



● Unsorted
○ Sorted

```
33
29
45
19
38
26
86
70
6
73
68
36
90
58
76
88
3
37
15
17
71
74
41
53
65
```

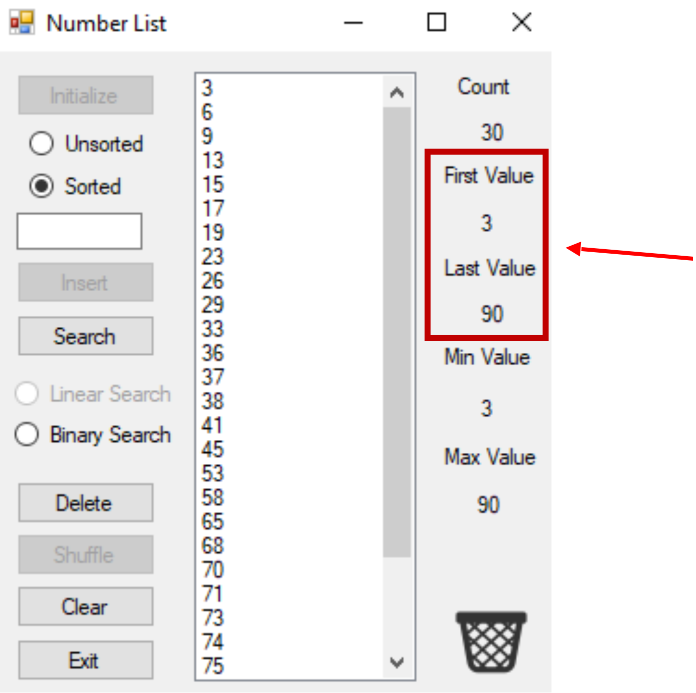
Count
30
First Value
33
Last Value
13
Min Value
3
Max Value
90

List contains 30 items, therefore count equals 30.

First and last values in the list are 33 and 13.

Min and Max values in the list are 3 and 90.

These stats change at runtime whenever the list updates even if the current list items are just sorted in ascending order for example. This would change the stats of first and last value whilst the count and min/max values would stay the same. Example shown below.



○ Unsorted
● Sorted

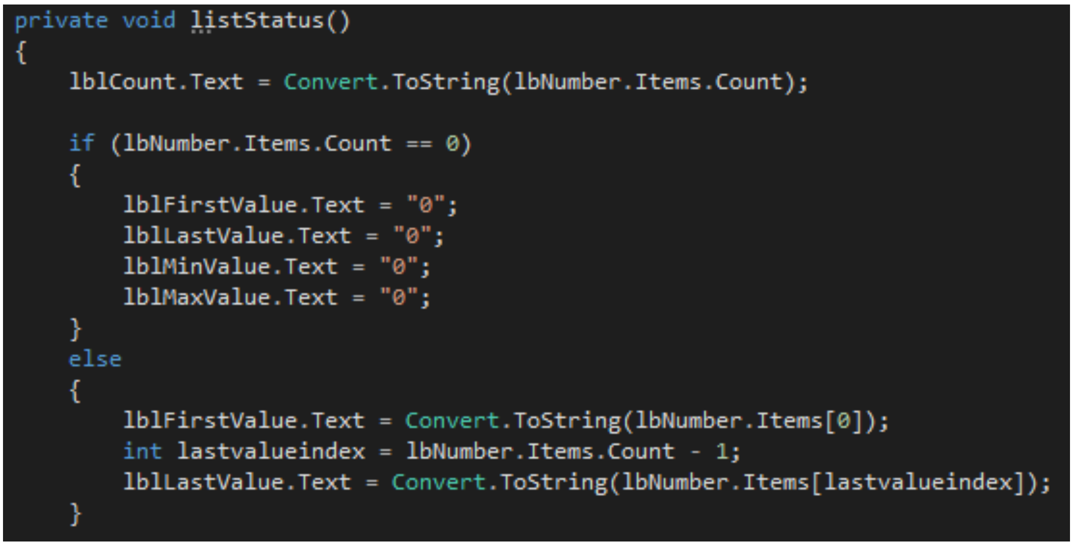
3
6
9
13
15
17
19
23
26
29
33
36
37
38
41
45
53
58
65
68
70
71
73
74
75

Count
30
First Value
3
Last Value
90
Min Value
3
Max Value
90

Count, Min/Max value stay the same after sorting in ascending order.

Values highlighted changed as 3 is now the first value in the list and 90 is the last value in the list.

‘listStatus’ used if else statements and for loops to find and calculate the statistics for each label. If the list was empty, then all values such as count, or last value would be equal to 0 as there are no items within the list. Else, the program would calculate the values of the first and last values contained in the list.

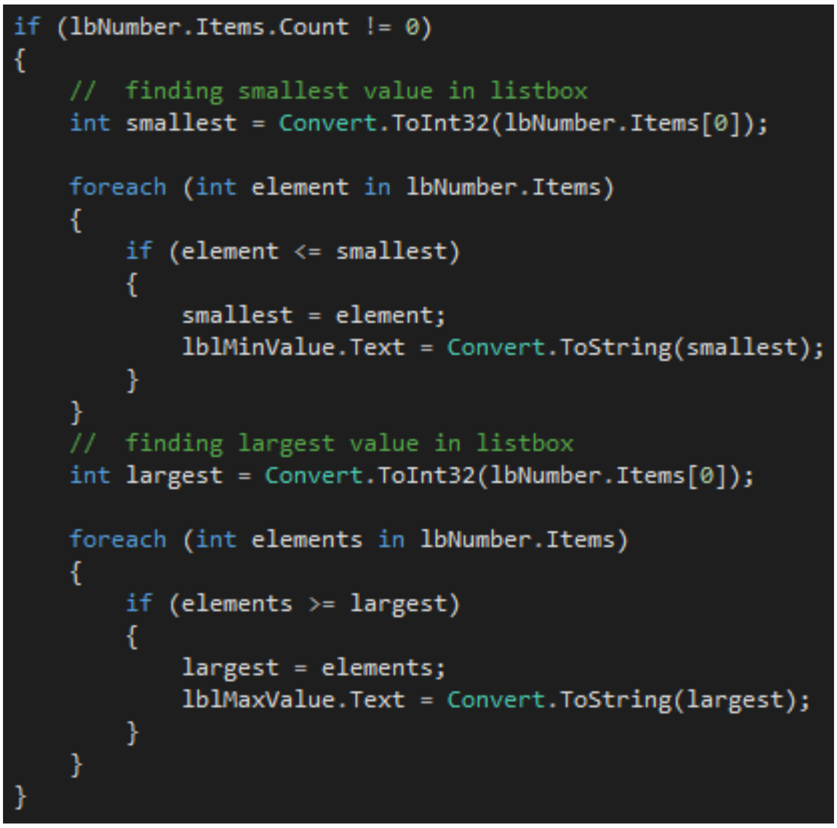


```
private void listStatus()
{
    lblCount.Text = Convert.ToString(lbNumber.Items.Count);

    if (lbNumber.Items.Count == 0)
    {
        lblFirstValue.Text = "0";
        lblLastValue.Text = "0";
        lblMinValue.Text = "0";
        lblMaxValue.Text = "0";
    }
    else
    {
        lblFirstValue.Text = Convert.ToString(lbNumber.Items[0]);
        int lastvalueindex = lbNumber.Items.Count - 1;
        lblLastValue.Text = Convert.ToString(lbNumber.Items[lastvalueindex]);
    }
```

If the list size is not equal to 0, then the smallest and largest value will be found via a for loop which checks each element in the list.

To find the smallest value in the list, the smallest variable is assigned the value of the first item in the list and checks if the element is smaller than or equal to the current smallest value. If so, the smallest variable is assigned the value stored in the element variable.

To find the largest value in the list, the largest variable is assigned the value of the first item in the list and checks if the element is larger than or equal to the current largest value. If so, the largest variable is assigned the value stored in the elements variable.

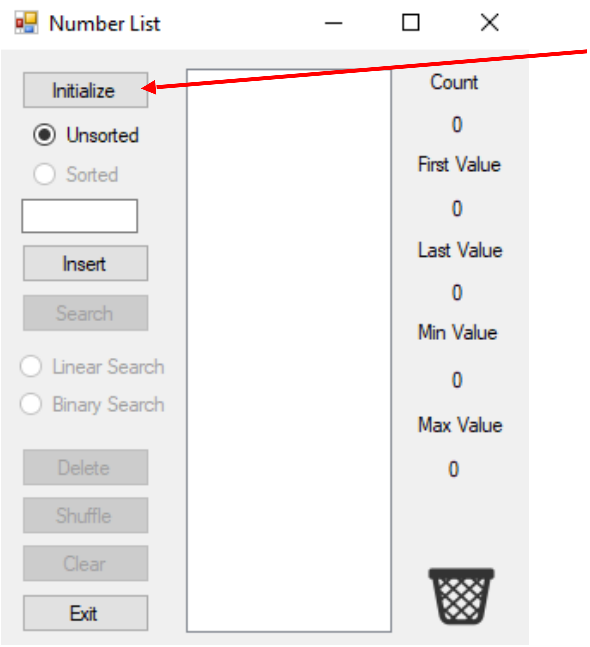


```
if (lbNumber.Items.Count != 0)
{
    //  finding smallest value in listbox
    int smallest = Convert.ToInt32(lbNumber.Items[0]);

    foreach (int element in lbNumber.Items)
    {
        if (element <= smallest)
        {
            smallest = element;
            lblMinValue.Text = Convert.ToString(smallest);
        }
    }
    //  finding largest value in listbox
    int largest = Convert.ToInt32(lbNumber.Items[0]);

    foreach (int elements in lbNumber.Items)
    {
        if (elements >= largest)
        {
            largest = elements;
            lblMaxValue.Text = Convert.ToString(largest);
        }
    }
}
```

Initialize

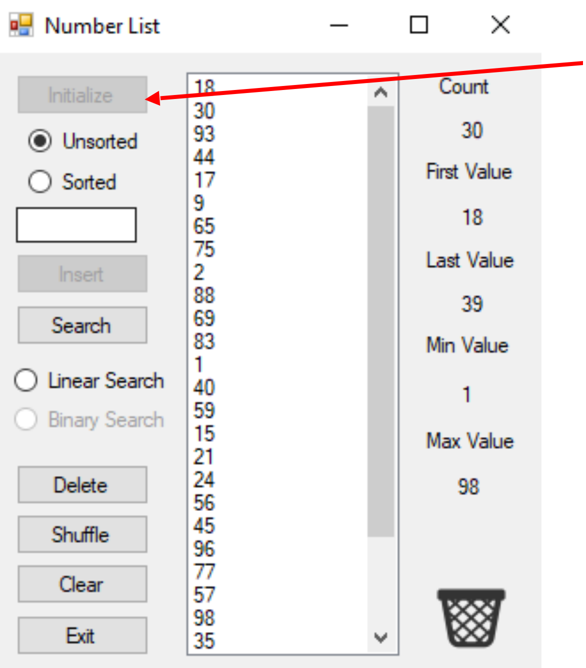
The 'Initialize' button named 'btnInitialize' fills the listbox with 30 unique random integer values ranging from 0-100. If the button is used when there are existing elements contained within the listbox, it will only add new values until the max count of 30 is reached. For example, if the listbox already contained 16 values, the initialize button will only add 14 new values.

Before



'Initialize' Button

After



● Unsorted

18
30
93
44
17
9
65
75
2
88
69
83
1
40
59
15
21
24
56
45
96
77
57
98
35

Count

30

First Value

18

Last Value

39

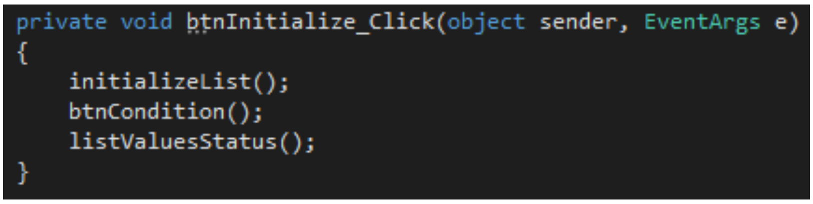
Min Value

1

Max Value

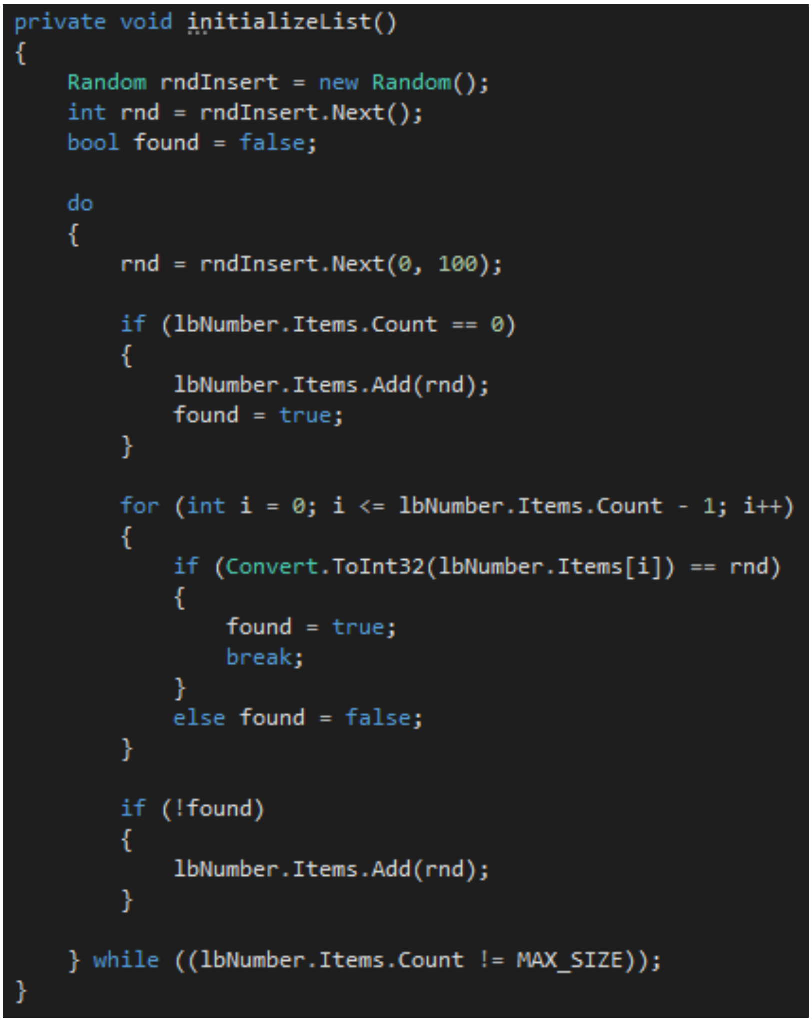
98

‘Initialize’ Button no longer accessible due to the listbox having a max value of 30 items



```
private void btnInitialize_Click(object sender, EventArgs e)
{
    initializeList();
    btnCondition();
    listValuesStatus();
}
```

Once the btnInitialize method is executed via a click event, it calls another method named ‘initializeList’. It was possible to contain this method within an if statement to prevent it from being executed when the listbox has the maximum value. However, this was not needed as the button will be disabled at runtime if the listbox is full.



```
private void initializeList()
{
    Random rndInsert = new Random();
    int rnd = rndInsert.Next();
    bool found = false;

    do
    {
        rnd = rndInsert.Next(0, 100);

        if (lbNumber.Items.Count == 0)
        {
            lbNumber.Items.Add(rnd);
            found = true;
        }

        for (int i = 0; i <= lbNumber.Items.Count - 1; i++)
        {
            if (Convert.ToInt32(lbNumber.Items[i]) == rnd)
            {
                found = true;
                break;
            }
            else found = false;
        }

        if (!found)
        {
            lbNumber.Items.Add(rnd);
        }

    } while ((lbNumber.Items.Count != MAX_SIZE));
}
```

Within ‘initializeList’, it obtains a random unique integer value using the built-in Random class from the System namespace. It does this via a do while loop which runs until it meets its condition that the listbox doesn’t contain 30 (MAX_SIZE) elements.

To prevent any duplicate values from being entered to the list, a for loop searches through the current contents until it finds a duplicate value which changes the bool variable ‘found’ to true which skips adding that specific value in the if statement (!found) and the loop starts again until the list reaches the maximum size. If the for loop doesn’t find a duplicate value, ‘found’ is set to false which then adds value stored in ‘rnd’ to the list.

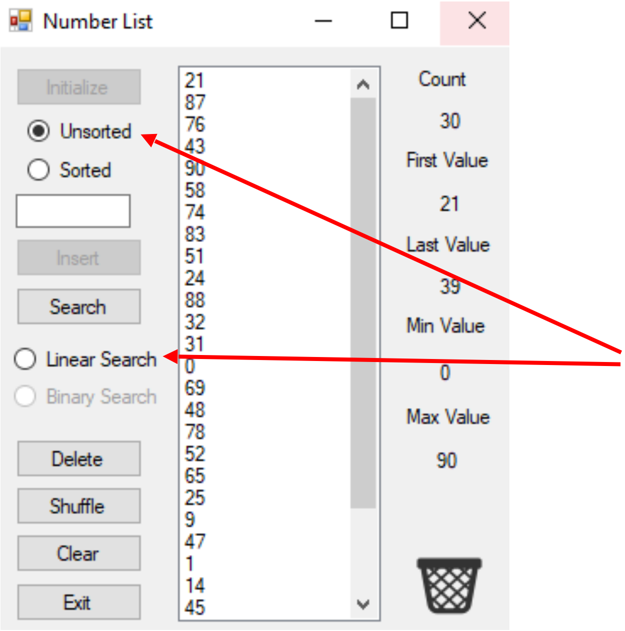
Once ‘initializeList’ finishes adding elements to the list, ‘btnCondition’ and ‘listStatus’ will then execute enabling/disabling certain controls and updating the new stats for count etc.

rbUnsorted & rbSorted

‘rbUnsorted’ and ‘rbSorted’ are used to determine how the list will be manipulated. ‘rbUnsorted’ is used to access functionality such as an unsorted initialized list, inserting any values to the end (top) of the list and the ability to use a linear search to find a specific value within the list.

‘rbSorted’ is used to access functionality which will sort the list in ascending order. Using ‘insert’, it will ensure that the inserted value is placed in order and it also allows the ability to use a binary search to find a specific value within the list.

Unsorted



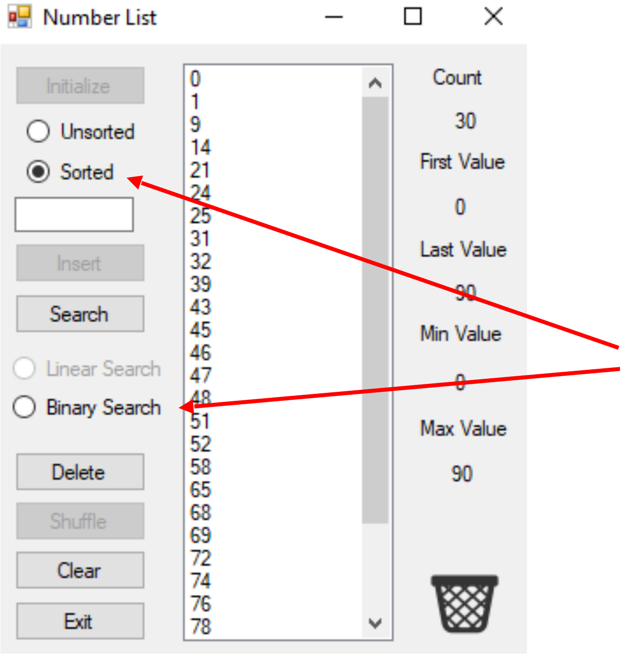
● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

```
21
87
76
43
90
58
74
83
51
24
88
32
31
0
69
48
78
52
65
25
9
47
1
14
45
```

Count
30
First Value
21
Last Value
39
Min Value
0
Max Value
90

‘Unsorted’ is checked therefore options such as ‘Linear Search’ is enabled

Sorted



○ Unsorted
● Sorted
○ Linear Search
○ Binary Search

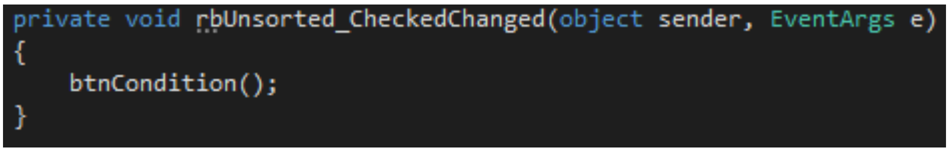
```
0
1
9
14
21
24
25
31
32
39
43
45
46
47
48
51
52
58
65
68
69
72
74
76
78
```

Count
30
First Value
0
Last Value
90
Min Value
0
Max Value
90

‘Sorted’ is checked therefore options such as ‘Binary Search’ is enabled. Also, the list has been sorted in ascending order

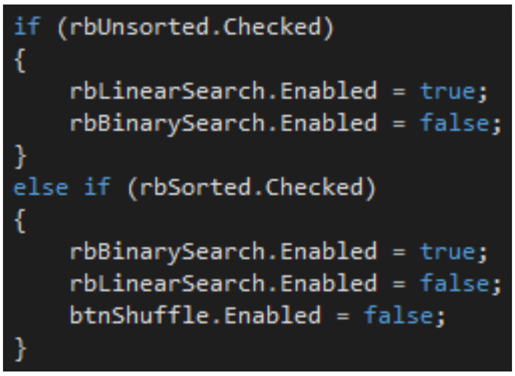
‘rbUnsorted_CheckChanged’ updates every control related to ‘rbUnsorted’ within ‘btnCondition’ allowing the functionality explained above. There is no need for ‘listStatus’ as ‘rbUnsorted_CheckChanged’ will never modify the list in any way.

Also, this applies for shuffling the order of items within the list as there’s already a dedicated control named ‘btnShuffle’ which has this functionality. ‘btnShuffle’ can only be used when ‘rbUnsorted’ is checked however.



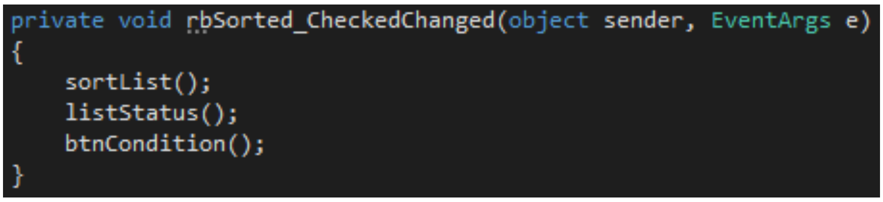
```
private void rbUnsorted_CheckedChanged(object sender, EventArgs e)
{
    btnCondition();
}
```

Code from ‘btnCondition’



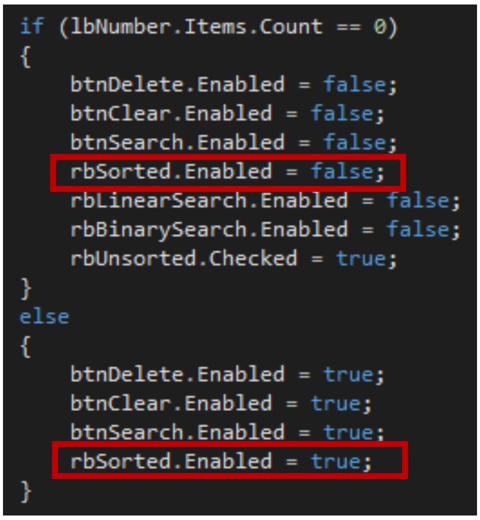
```
if (rbUnsorted.Checked)
{
    rbLinearSearch.Enabled = true;
    rbBinarySearch.Enabled = false;
}
else if (rbSorted.Checked)
{
    rbBinarySearch.Enabled = true;
    rbLinearSearch.Enabled = false;
    btnShuffle.Enabled = false;
}
```

‘rbSorted_CheckChanged’ can only be used when the list is not empty as it’s not possible to sort an empty list and this can lead to the program crashing or unexpected bugs. However, due to built-in code within ‘btnCondition’, an if statement to check for errors within ‘rbSorted_CheckChanged’ because ‘rbSorted’ will only enable if the list isn’t empty and only disable when the list is empty.



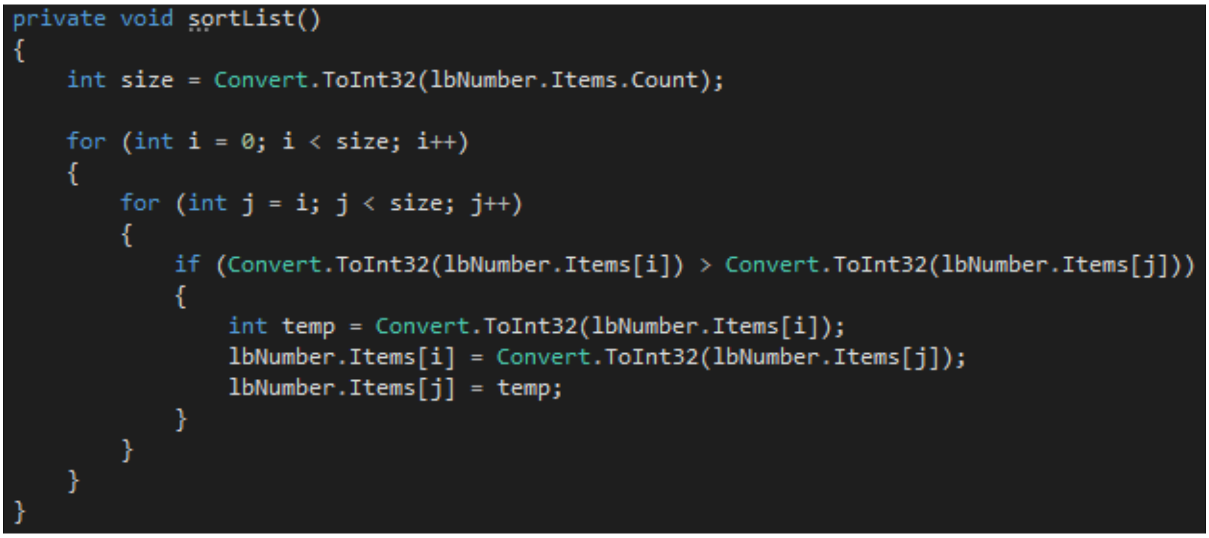
```
private void rbSorted_CheckedChanged(object sender, EventArgs e)
{
    sortList();
    listStatus();
    btnCondition();
}
```

More code from ‘btnCondition’



```
if (lbNumber.Items.Count == 0)
{
    btnDelete.Enabled = false;
    btnClear.Enabled = false;
    btnSearch.Enabled = false;
    rbSorted.Enabled = false;
    rbLinearSearch.Enabled = false;
    rbBinarySearch.Enabled = false;
    rbUnsorted.Checked = true;
}
else
{
    btnDelete.Enabled = true;
    btnClear.Enabled = true;
    btnSearch.Enabled = true;
    rbSorted.Enabled = true;
}
```

‘rbSorted_CheckChanged’ calls a method named ‘sortList’ which sorts the list into ascending order. To achieve this, I used a sorting algorithm called bubble sort which essentially works by swapping adjacent elements if they’re in the wrong order. As the sort is focusing on ascending order, it checks if the current value is greater than the adjacent value. If true, it swaps the two elements and moves onto the next check. If false, it skips the current element and checks the next.



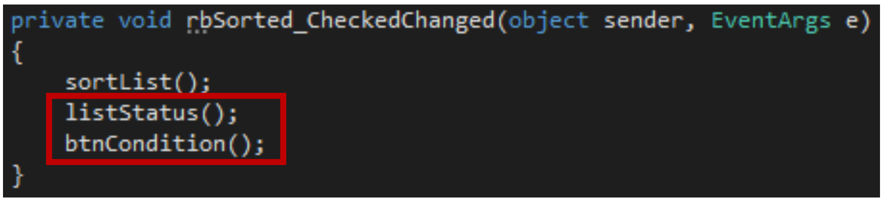
```
private void sortList()
{
    int size = Convert.ToInt32(lbNumber.Items.Count);

    for (int i = 0; i < size; i++)
    {
        for (int j = i; j < size; j++)
        {
            if (Convert.ToInt32(lbNumber.Items[i]) > Convert.ToInt32(lbNumber.Items[j]))
            {
                int temp = Convert.ToInt32(lbNumber.Items[i]);
                lbNumber.Items[i] = Convert.ToInt32(lbNumber.Items[j]);
                lbNumber.Items[j] = temp;
            }
        }
    }
}
```

The ‘size’ variable holds the current total size of the list which is used to indicate how many times the for loops iterate the swap which checks each element within the list.

As items[j] is always i+1, this essentially works the same as checking the adjacent element e.g. items[i (4)] > items[i (5)] etc.

If the value in items[i] is greater than the value in the adjacent items[j], then the swap takes place assigning the variable ‘temp’ the value of items[i] to make the swap. Then, items[i] receives the value in items[j]. Therefore items[j] finally receives the value in temp (previously items[i]) meaning that the swap has finally taken place.

Once the sort is complete, ‘rbSorted_CheckChanged’ updates the stats (First/Last values) and enables/disables the required controls.

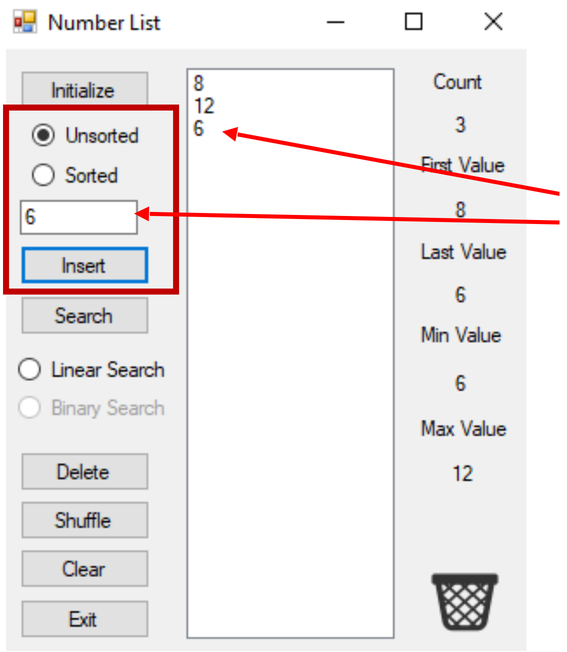


```
private void rbSorted_CheckedChanged(object sender, EventArgs e)
{
    sortList();
    listStatus();
    btnCondition();
}
```

Insert

Insert is used to insert a non-duplicate integer value in the range 0-100. Depending on which radio button is checked for unsorted and sort, the insert method will either insert a value at the top (end) of the list or insert a value in the correct position for ascending order.

Unsorted Insert



Number List

● Unsorted
○ Sorted

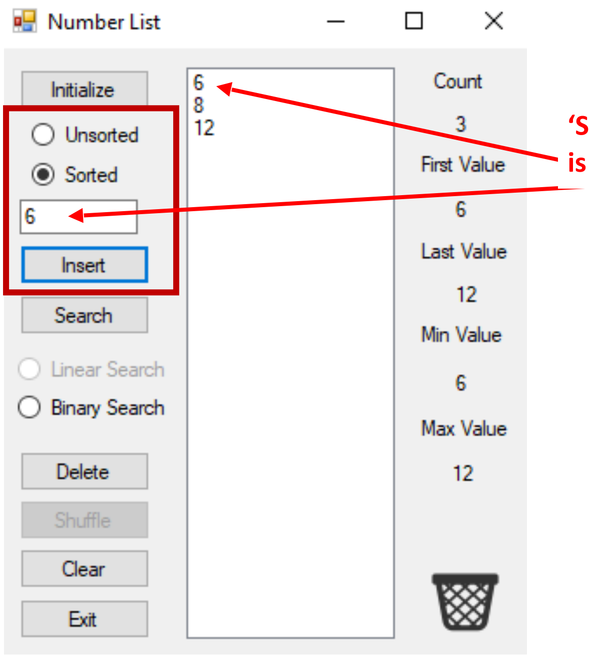
6

```
8
12
6
```

Count
3
First Value
8
Last Value
6
Min Value
6
Max Value
12

‘Unsorted’ is checked therefore element is inserted at the end of the list

Sorted Insert



Number List

○ Unsorted
● Sorted

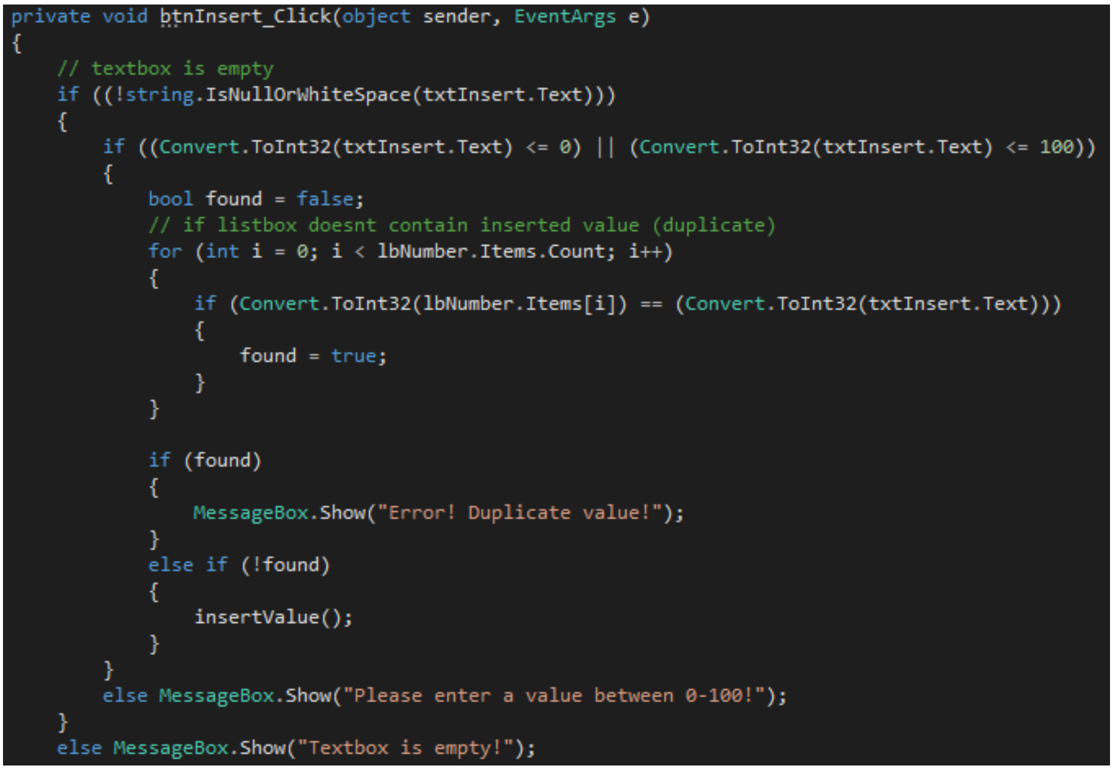
6

```
6
8
12
```

Count
3
First Value
6
Last Value
12
Min Value
6
Max Value
12

‘Sorted’ is checked therefore element is inserted in ascending order

‘btnInsert_Click’ uses an if else statement to catch any errors such as entering a duplicate value or attempting to insert a null value (empty textbox) for example.

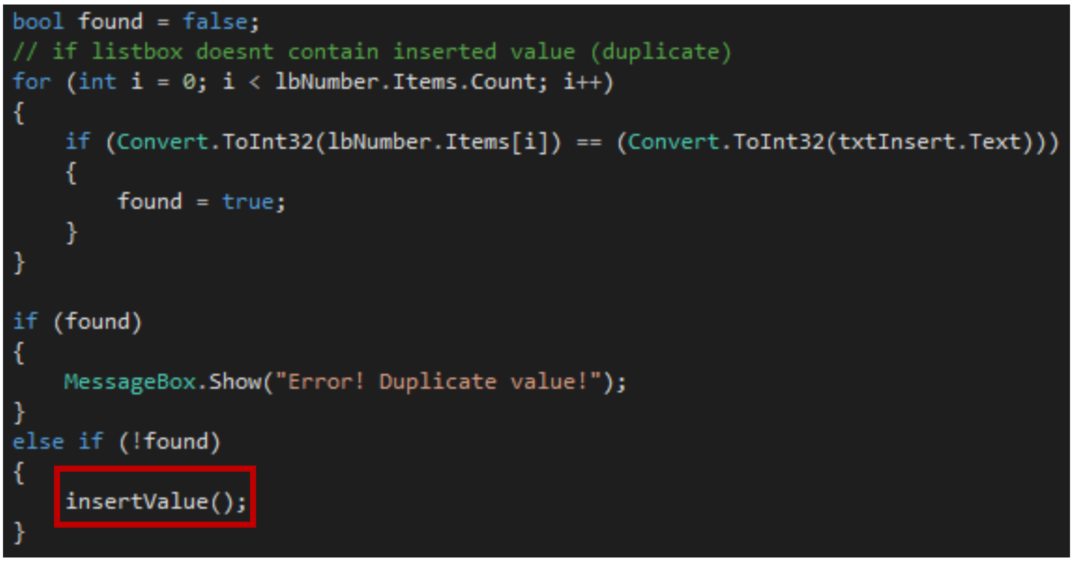


```
private void btnInsert_Click(object sender, EventArgs e)
{
    // textbox is empty
    if ((!string.IsNullOrWhiteSpace(txtInsert.Text)))
    {
        if ((Convert.ToInt32(txtInsert.Text) <= 0) || (Convert.ToInt32(txtInsert.Text) <= 100))
        {
            bool found = false;
            // if listbox doesnt contain inserted value (duplicate)
            for (int i = 0; i < lbNumber.Items.Count; i++)
            {
                if (Convert.ToInt32(lbNumber.Items[i]) == (Convert.ToInt32(txtInsert.Text)))
                {
                    found = true;
                }
            }

            if (found)
            {
                MessageBox.Show("Error! Duplicate value!");
            }
            else if (!found)
            {
                insertValue();
            }
        }
        else MessageBox.Show("Please enter a value between 0-100!");
    }
    else MessageBox.Show("Textbox is empty!");
```

First, it checks if the textbox is not null before allowing any comparisons to happen within the conditions as this would lead to exceptions happening. If the textbox is not null, it checks if the value entered is between the range 0-100.

Then, it uses a bool variable ‘found’ set to false which will be used as a condition further on in the if statement. A for loop is then used that checks for duplicate values by searching the entire current contents of the list. If a duplicate is found, ‘found’ will be set to true which displays an error message stating that you’re attempting to insert a duplicate value.

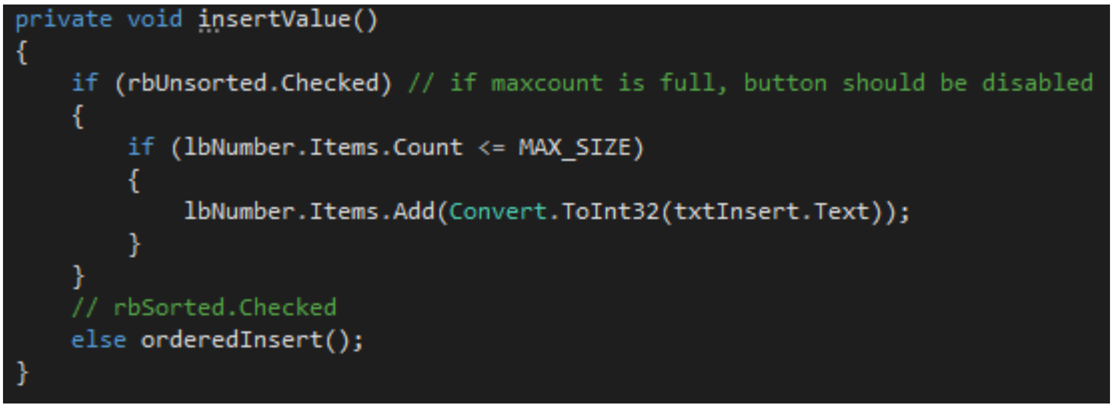
If a duplicate value is not found, ‘found’ stays false and calls the ‘insertValue’ method which contains the code necessary to insert values.



```
bool found = false;
// if listbox doesnt contain inserted value (duplicate)
for (int i = 0; i < lbNumber.Items.Count; i++)
{
    if (Convert.ToInt32(lbNumber.Items[i]) == (Convert.ToInt32(txtInsert.Text)))
    {
        found = true;
    }
}

if (found)
{
    MessageBox.Show("Error! Duplicate value!");
}
else if (!found)
{
    insertValue();
}
```

‘insertValue’ checks if either ‘rbUnsorted’ or ‘rbSorted’ is checked before selecting an insertion method. If ‘rbUnsorted’ is checked, it inserts a value to the end of the list. There is no need for a conditional statement checking if list isn't full as 'Insert' is disabled by 'btnCondition' when the list is full. This also applies for the 'else orderedInsert' as there are only two radio buttons unsorted and sorted.

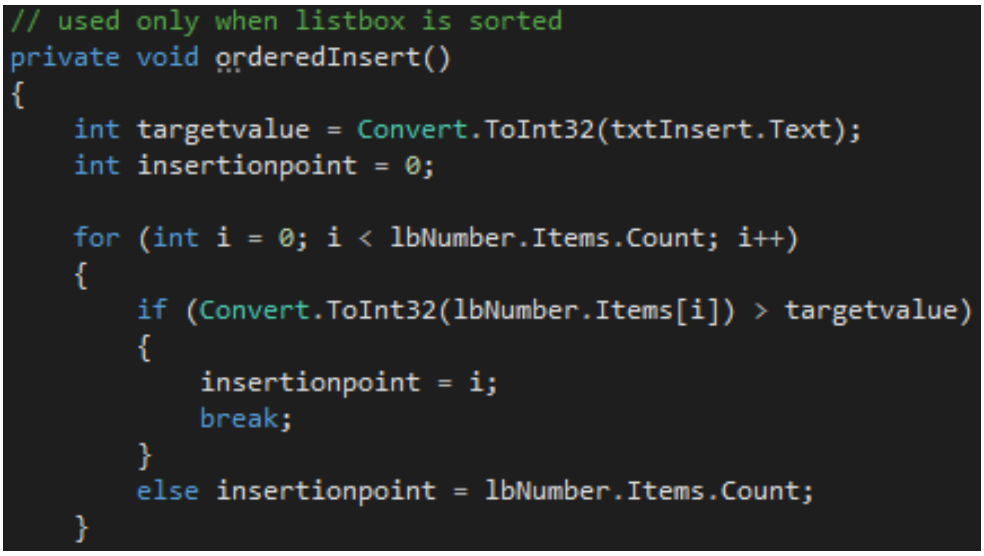
Else if 'rbSorted.Checked', it calls the method 'orderedInsert' which inserts values in order.



```
private void insertValue()
{
    if (rbUnsorted.Checked) // if maxcount is full, button should be disabled
    {
        if (lbNumber.Items.Count <= MAX_SIZE)
        {
            lbNumber.Items.Add(Convert.ToInt32(txtInsert.Text));
        }
    }
    // rbSorted.Checked
    else orderedInsert();
}
```

'orderedInsert' works by using a for loop by searching the entire sorted list until iterating until it reaches the condition that the value contained within the specified index is greater than the target value. Then, the insertion point is assigned the positional index of the list using 'i' allowing us to insert a value into specified index.

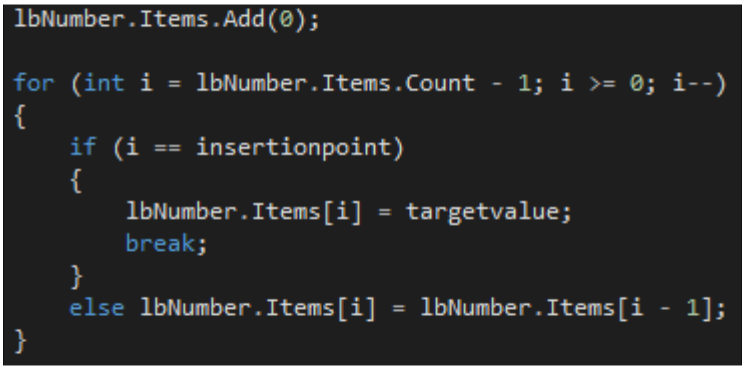
If the target value is greater than the entirety of the sorted list, it is assigned the position of the final index of the list.



```
// used only when listbox is sorted
private void orderedInsert()
{
    int targetvalue = Convert.ToInt32(txtInsert.Text);
    int insertionpoint = 0;

    for (int i = 0; i < lbNumber.Items.Count; i++)
    {
        if (Convert.ToInt32(lbNumber.Items[i]) > targetvalue)
        {
            insertionpoint = i;
            break;
        }
        else insertionpoint = lbNumber.Items.Count;
    }
```

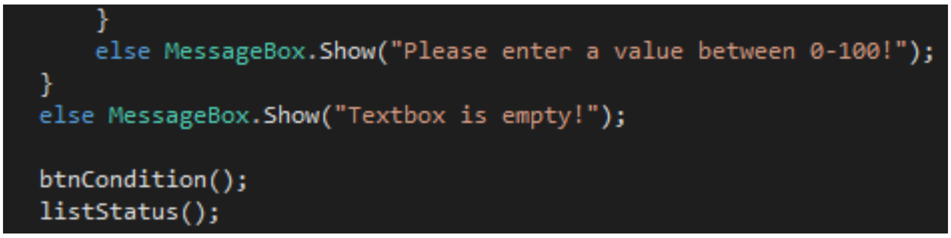
The list then adds a new element to the end of the list for us to shift the values greater than the insertion point up to be able to insert the new element.

A for loop is then used to search the list from the end of the list to the start by decrementing until it finds the insertion point. The else statement shifts the items within the list in order to make space for the inserted value. Once it finds the insertion point, the value contained within ‘targetvalue’ will be assigned to the list completing the ordered insert.



```
lbNumber.Items.Add(0);

for (int i = lbNumber.Items.Count - 1; i >= 0; i--)
{
    if (i == insertionpoint)
    {
        lbNumber.Items[i] = targetvalue;
        break;
    }
    else lbNumber.Items[i] = lbNumber.Items[i - 1];
}
```

As shown before, ‘btnCondition’ and ‘listStatus’ are called at the end of the method updating the controls and stats needed.



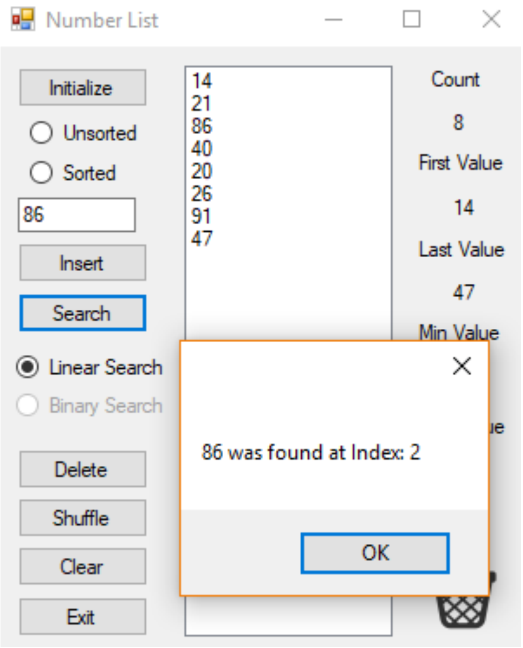
```
      }
      else MessageBox.Show("Please enter a value between 0-100!");
  }
  else MessageBox.Show("Textbox is empty!");

  btnCondition();
  listStatus();
```

Search

Search allows the user to search the entire list for the specified value inputted by the user. There are two search options called linear search and binary search which are only enabled when their respective unsorted or sorted radio buttons are checked.

Unsorted allows the use of linear search and sorted allows the use of binary search. This is because linear search is better equipped for unsorted lists whereas binary search is better for sorted lists.

Unsorted Linear Search



○ Unsorted
○ Sorted

86

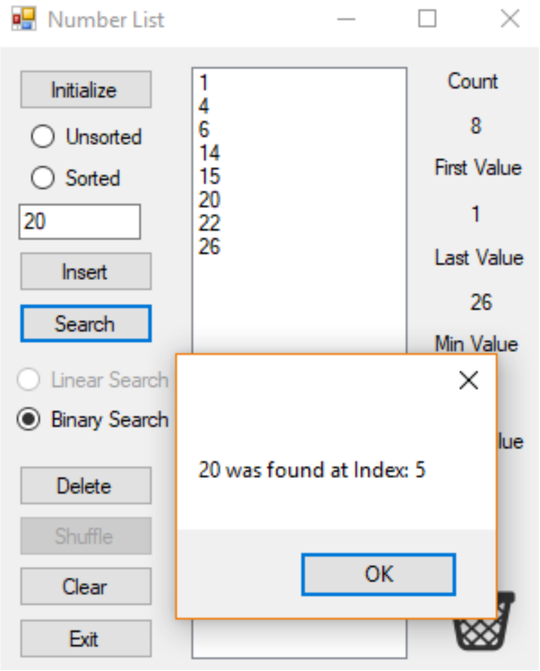
● Linear Search
○ Binary Search

14
21
86
40
20
26
91
47

Count
8
First Value
14
Last Value
47
Min Value

86 was found at Index: 2

Sorted Binary Search



○ Unsorted
○ Sorted

20

○ Linear Search
● Binary Search

1
4
6
14
15
20
22
26

Count
8
First Value
1
Last Value
26
Min Value

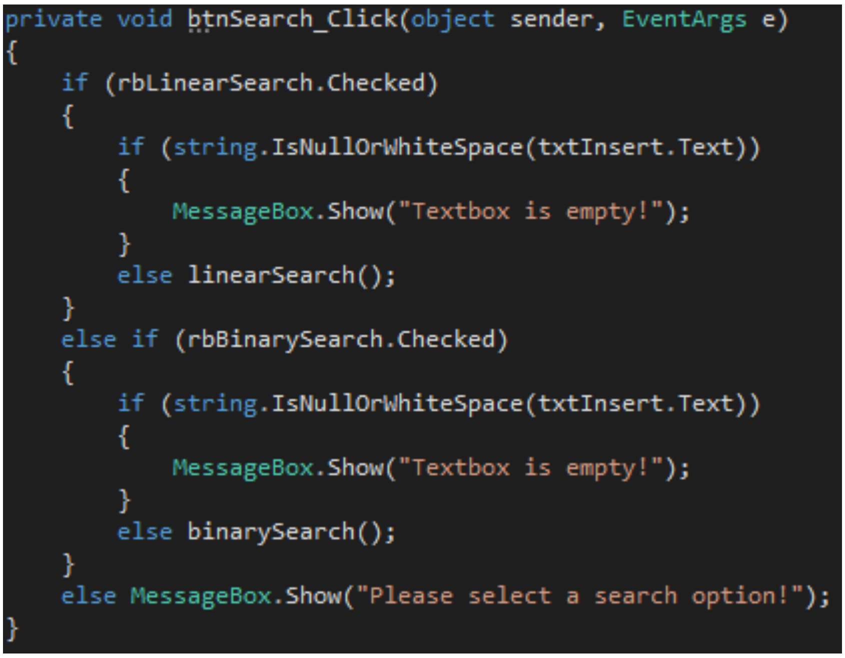
20 was found at Index: 5

'btnSearch_Click' uses if else statements to specify which search method should be used checks if the textbox is null (empty).

If ‘rbLinearSearch’ is checked, it selects ‘linearSearch’.

If ‘rbBinarySearch’ is checked, it selects ‘binarySearch’.

If no radio button is checked, it shows a message prompt requesting the user to select a search option.



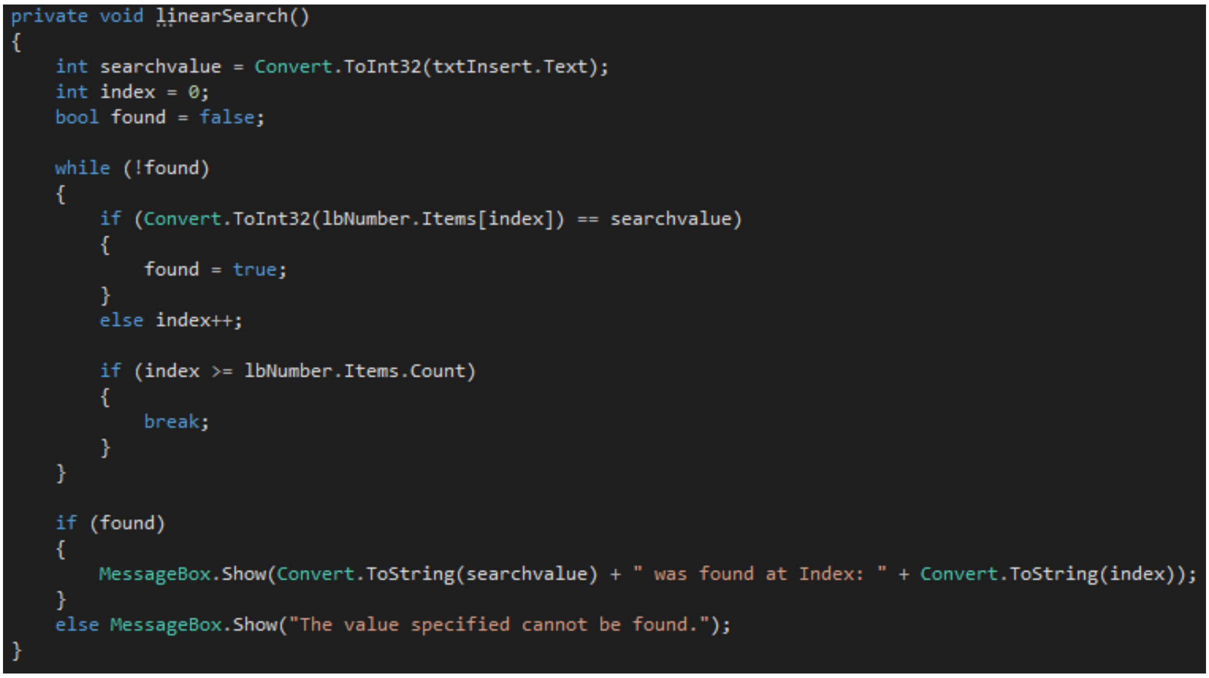
```
private void btnSearch_Click(object sender, EventArgs e)
{
    if (rbLinearSearch.Checked)
    {
        if (string.IsNullOrWhiteSpace(txtInsert.Text))
        {
            MessageBox.Show("Textbox is empty!");
        }
        else linearSearch();
    }
    else if (rbBinarySearch.Checked)
    {
        if (string.IsNullOrWhiteSpace(txtInsert.Text))
        {
            MessageBox.Show("Textbox is empty!");
        }
        else binarySearch();
    }
    else MessageBox.Show("Please select a search option!");
}
```

Linear Search

Linear search checks each element in the list sequentially until a match is found or until all elements within the list have been searched.

‘linearSearch’ uses a bool variable ‘found’ that keeps looping in a while loop until a match has been found. If there is a match, ‘found’ will be set to true else it will increment and start again. If the index increment overtakes the max count of the list, the while loop breaks to stop the program from continuously looping.

If found is true, the search value and position (index) will be displayed via message prompt. Else it will specify that the value cannot be found.



```
private void linearSearch()
{
    int searchvalue = Convert.ToInt32(txtInsert.Text);
    int index = 0;
    bool found = false;

    while (!found)
    {
        if (Convert.ToInt32(lbNumber.Items[index]) == searchvalue)
        {
            found = true;
        }
        else index++;

        if (index >= lbNumber.Items.Count)
        {
            break;
        }
    }

    if (found)
    {
        MessageBox.Show(Convert.ToString(searchvalue) + " was found at Index: " + Convert.ToString(index));
    }
    else MessageBox.Show("The value specified cannot be found.");
}
```

Binary Search

Binary searches are used on lists that have already been sorted. This is because binary searches work by dividing the search interval in half. The first search interval covers the whole list and checks if the target value is less or greater than the value in the middle of the interval. If the value is lower, it narrows the search interval to the lower half and if the value is higher, it narrows the search interval to the upper half. The search interval then does the same process with the lower or higher half dividing it by half.

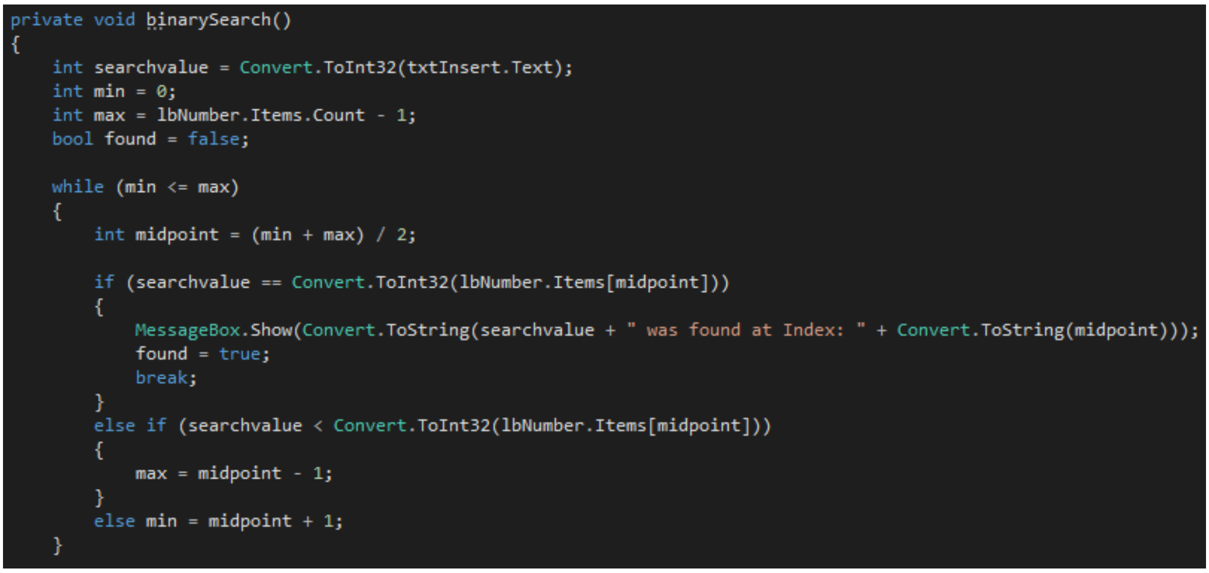
This process repeats until the value is found or the search interval is empty.

‘binarySearch’ works by using a while loop that loops when the min count value is less than or equal to the max count value.

The while loop will divide the min + max by 2 to obtain the midpoint. If the search value is equal to the midpoint, a message prompt displays the search value was found at the specified index. The bool variable ‘found’ is then set to true. It then breaks out of loop.

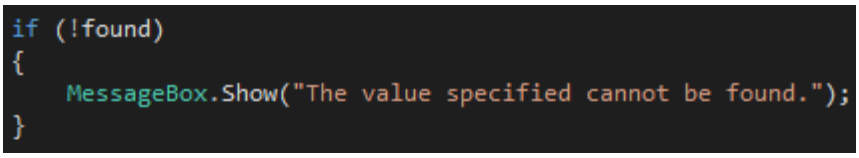
Else if the search value is less than the midpoint, the max count value becomes the midpoint – 1. Else the min count value becomes the midpoint + 1.

This process repeats until the value is found.



```
private void binarySearch()
{
    int searchvalue = Convert.ToInt32(txtInsert.Text);
    int min = 0;
    int max = lbNumber.Items.Count - 1;
    bool found = false;

    while (min <= max)
    {
        int midpoint = (min + max) / 2;

        if (searchvalue == Convert.ToInt32(lbNumber.Items[midpoint]))
        {
            MessageBox.Show(Convert.ToString(searchvalue + " was found at Index: " + Convert.ToString(midpoint)));
            found = true;
            break;
        }
        else if (searchvalue < Convert.ToInt32(lbNumber.Items[midpoint]))
        {
            max = midpoint - 1;
        }
        else min = midpoint + 1;
    }
```

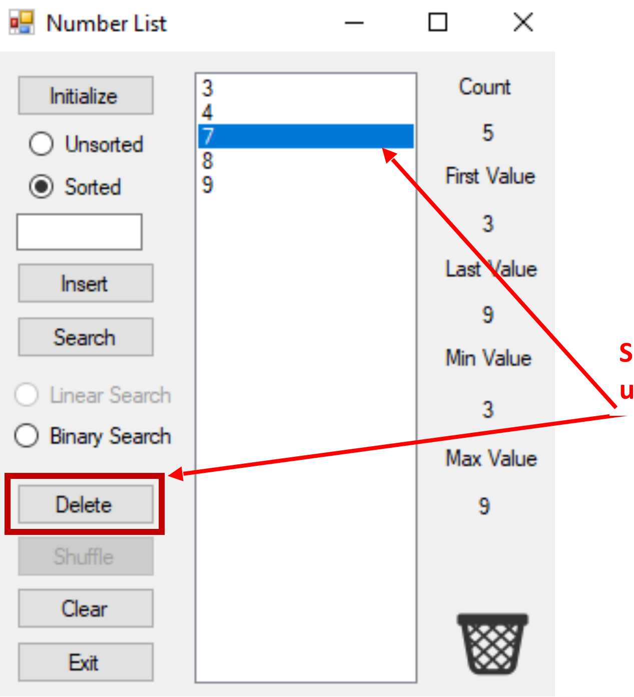
If the value is not found, a message prompt appears stating that “The value specified cannot be found”.



```
if (!found)
{
    MessageBox.Show("The value specified cannot be found.");
}
```

Delete

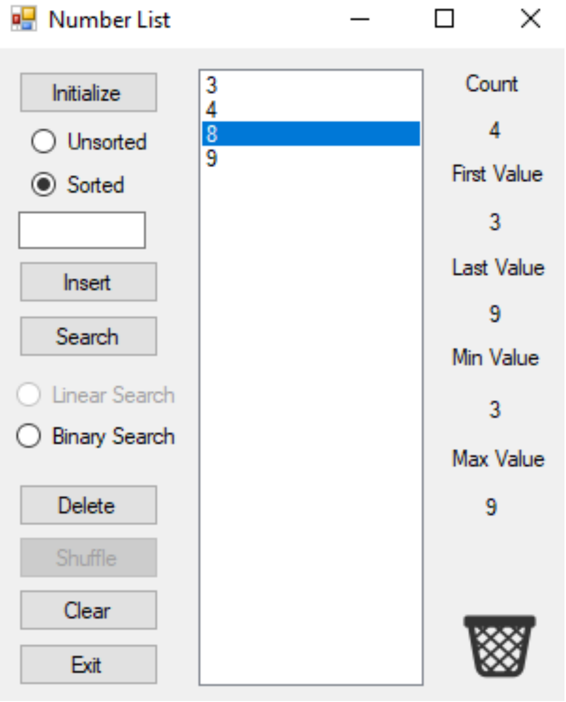
Delete allows the user to select an item from the list and remove (delete) it from the list.



3
4
7
8
9

Count
5
First Value
3
Last Value
9
Min Value
3
Max Value
9

Selected item is deleted from the list using the 'Delete' button



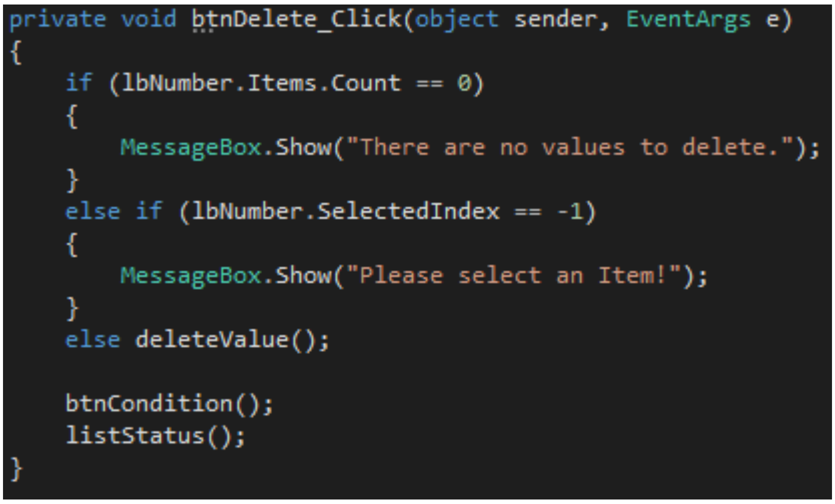
3
4
8
9

Count
4
First Value
3
Last Value
9
Min Value
3
Max Value
9

The value 7 no longer exists in the list

'btnDelete_Click' uses if else statements to catch errors before allowing an item to be deleted. If the list is empty, a message prompt will state that "There are no values to delete".

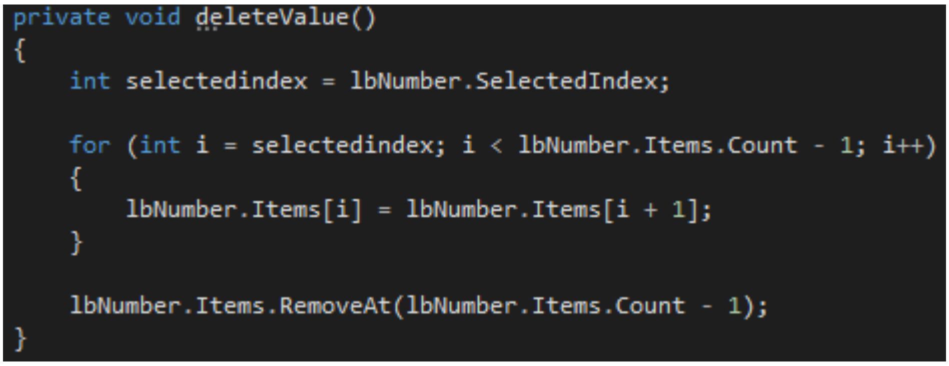
Else if no item is selected (SelectedIndex == -1 means no item selected), a message prompt will tell the user to select an item.



```
private void btnDelete_Click(object sender, EventArgs e)
{
    if (lbNumber.Items.Count == 0)
    {
        MessageBox.Show("There are no values to delete.");
    }
    else if (lbNumber.SelectedIndex == -1)
    {
        MessageBox.Show("Please select an Item!");
    }
    else deleteValue();

    btnCondition();
    listStatus();
}
```

'deleteValue' obtains the selectedindex to use in a for loop that iterates from the selectedindex to the total size of items in the list.

It shifts the entire list down so it's possible to remove the selected item without leaving an empty item behind.



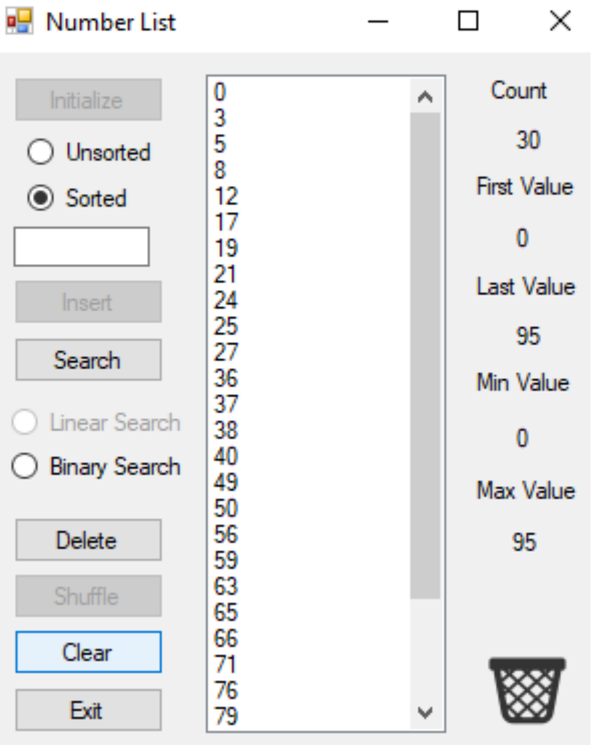
```
private void deleteValue()
{
    int selectedindex = lbNumber.SelectedIndex;

    for (int i = selectedindex; i < lbNumber.Items.Count - 1; i++)
    {
        lbNumber.Items[i] = lbNumber.Items[i + 1];
    }

    lbNumber.Items.RemoveAt(lbNumber.Items.Count - 1);
}
```

'btnCondition' and 'listStatus' are then called which updates the controls and stats for the program.

Shuffle

Shuffle jumbles every item in the list and is only accessible when 'rbUnsorted' is checked.

Before



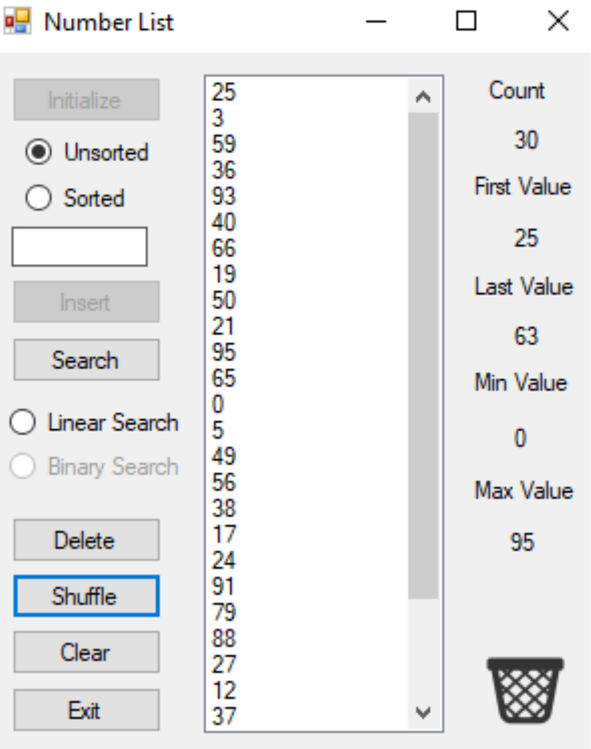
○ Unsorted
● Sorted

○ Linear Search
○ Binary Search

```
0
3
5
8
12
17
19
21
24
25
27
36
37
38
40
49
50
56
59
63
65
66
71
76
79
```

Count
30
First Value
0
Last Value
95
Min Value
0
Max Value
95

After



● Unsorted
○ Sorted

○ Linear Search
○ Binary Search

```
25
3
59
36
93
40
66
19
50
21
95
65
0
5
49
56
38
17
24
91
79
88
27
12
37
```

Count
30
First Value
25
Last Value
63
Min Value
0
Max Value
95

The shuffle works by obtaining random values from ‘rndShuffle’ which makes it possible to randomly change the order of the items in the list.

List.ObjectCollections list is an object that represents all items within the list. This makes it possible to access any item within the list to shuffle them.

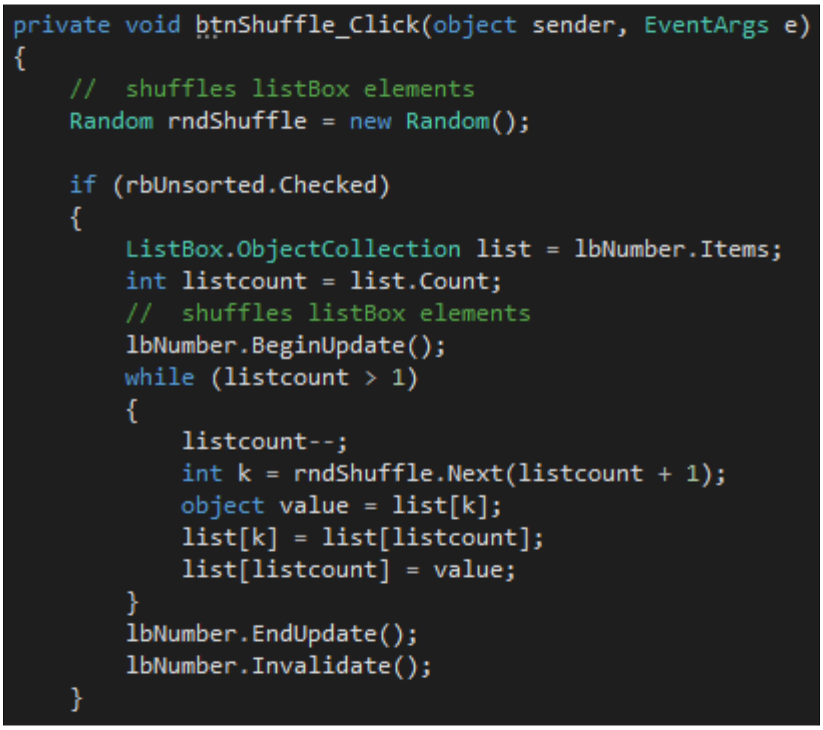
lbNumber.BeginUpdate, EndUpdate, Invalidate are used to prevent the listbox from drawing until the shuffle is complete.

The shuffle is done in a while loop which decrements until listcount is no longer greater than 1. This is because it will shuffle every single item once to make it random.

The variable ‘k’ then stores the random index obtained by rndShuffle.Next.

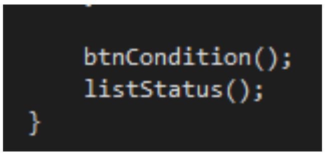
The value at the index location ‘k’ is then stored within the object ‘value’.

The items in the list are now shuffled after list[k] now obtains the location contained within listcount.



```
private void btnShuffle_Click(object sender, EventArgs e)
{
    //  shuffles listBox elements
    Random rndShuffle = new Random();

    if (rbUnsorted.Checked)
    {
        ListBox.ObjectCollection list = lbNumber.Items;
        int listcount = list.Count;
        //  shuffles listBox elements
        lbNumber.BeginUpdate();
        while (listcount > 1)
        {
            listcount--;
            int k = rndShuffle.Next(listcount + 1);
            object value = list[k];
            list[k] = list[listcount];
            list[listcount] = value;
        }
        lbNumber.EndUpdate();
        lbNumber.Invalidate();
    }
```

‘btnCondition’ and ‘listStatus’ are called to update the controls and stats for the program.

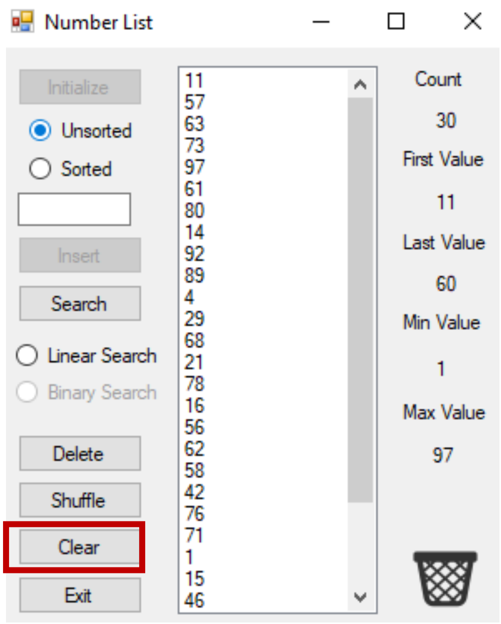


```
    btnCondition();
    listStatus();
}
```

Clear

Clear removes every element within the list making it empty.

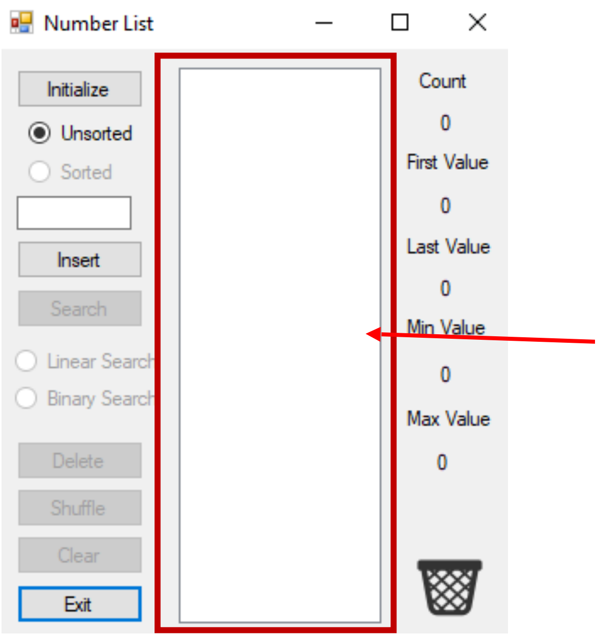
Before



```
11
57
63
73
97
61
80
14
92
89
4
29
68
21
78
16
56
62
58
42
76
71
1
15
46
```

Count: 30
First Value: 11
Last Value: 60
Min Value: 1
Max Value: 97

After

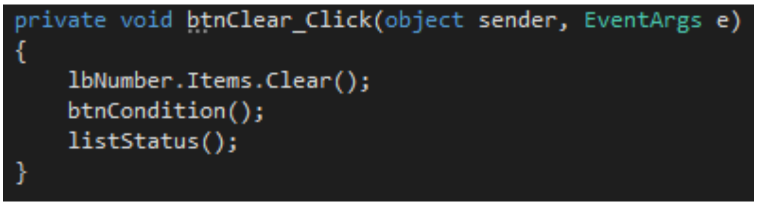


Count: 0
First Value: 0
Last Value: 0
Min Value: 0
Max Value: 0

List is empty after using the 'Clear' button

'btnClear_Click' uses '.Clear' which removes all items from the list 'lbNumber'.

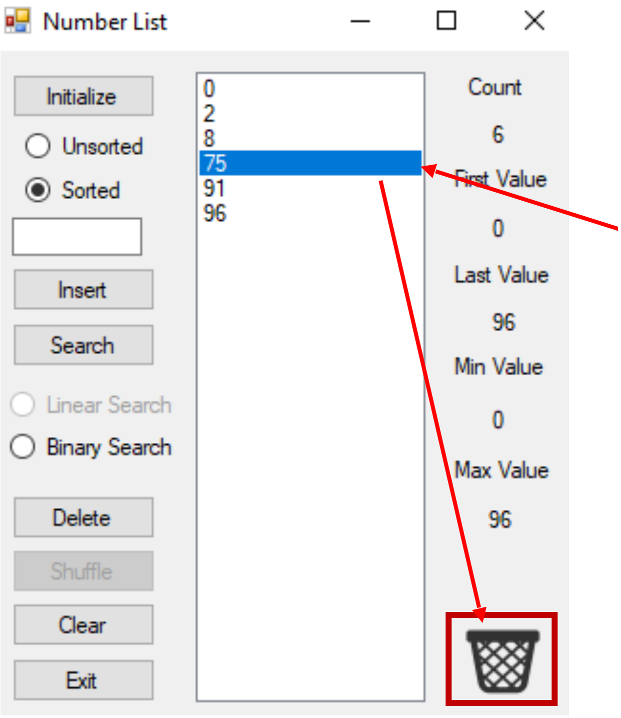
'btnCondition' and 'listStatus' then updates the controls and stats for the program.



```
private void btnClear_Click(object sender, EventArgs e)
{
    lbNumber.Items.Clear();
    btnCondition();
    listStatus();
}
```

Delete – Drag and Drop Icon

This allows the user to drag and drop any item from the list to a picture box which uses the same method of removing elements as the delete button.



○ Unsorted
● Sorted

0
2
8
75
91
96

Count 6

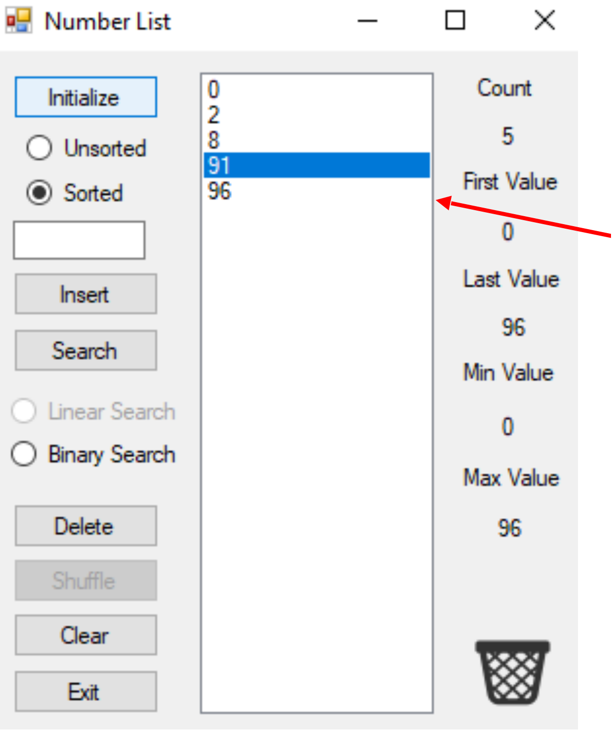
First Value 0

Last Value 96

Min Value 0

Max Value 96

Selected item is dragged to the Picture Box which uses a trash bin as an icon



○ Unsorted
● Sorted

0
2
8
91
96

Count 5

First Value 0

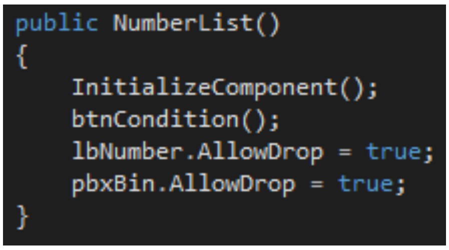
Last Value 96

Min Value 0

Max Value 96

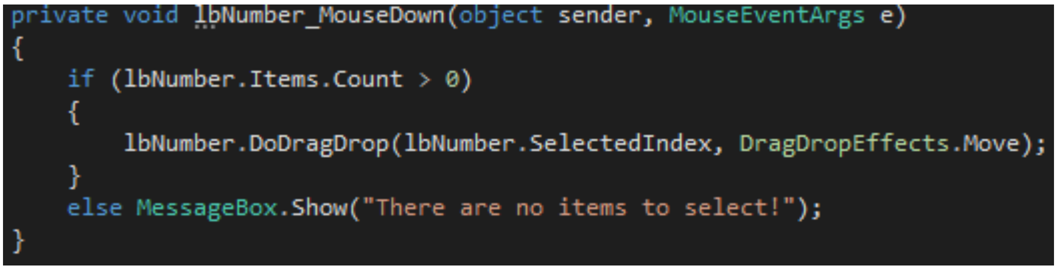
Item is now removed from the list

For the functionality of the drag and drop to work, ‘AllowDrop’ must be set to true.



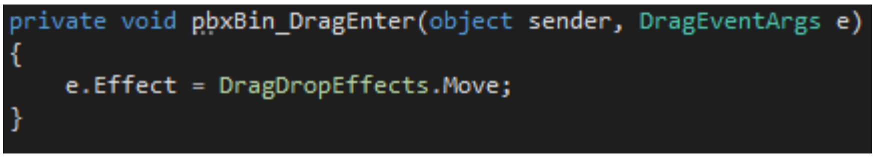
```
public NumberList()
{
    InitializeComponent();
    btnCondition();
    lbNumber.AllowDrop = true;
    pbxBin.AllowDrop = true;
}
```

‘lbNumber_MouseDown’ uses an if else statement that prevents the user from selecting an empty item from the list.

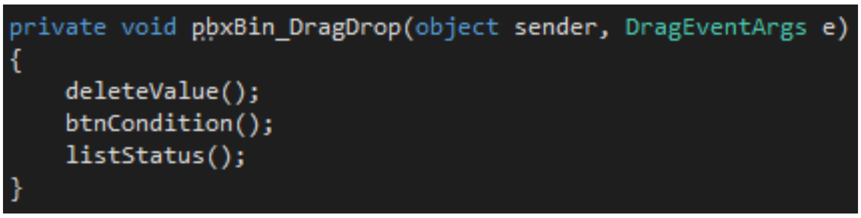


```
private void lbNumber_MouseDown(object sender, MouseEventArgs e)
{
    if (lbNumber.Items.Count > 0)
    {
        lbNumber.DoDragDrop(lbNumber.SelectedIndex, DragDropEffects.Move);
    }
    else MessageBox.Show("There are no items to select!");
}
```

The SelectedIndex carries over to ‘pbxBin’ and executes the ‘deleteValue’ method as seen above in the ‘Delete’ button functionality.



```
private void pbxBin_DragEnter(object sender, DragEventArgs e)
{
    e.Effect = DragDropEffects.Move;
}
```

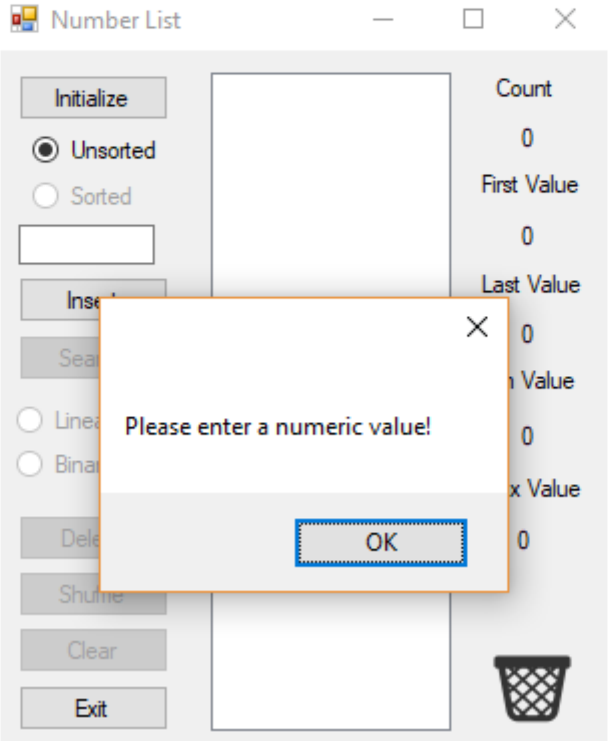


```
private void pbxBin_DragDrop(object sender, DragEventArgs e)
{
    deleteValue();
    btnCondition();
    listStatus();
}
```

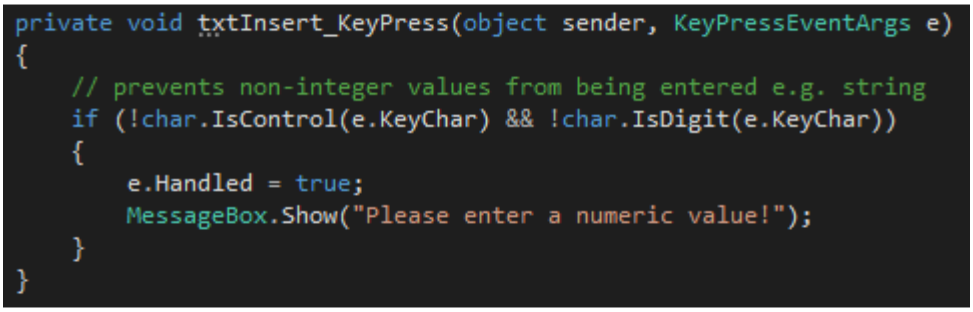
‘btnCondition’ and ‘listStatus’ then updates the controls and stats for the program.

Textbox

The textbox only accepts integer values rejecting any non-integer values which helps prevent errors such as users trying to input strings.



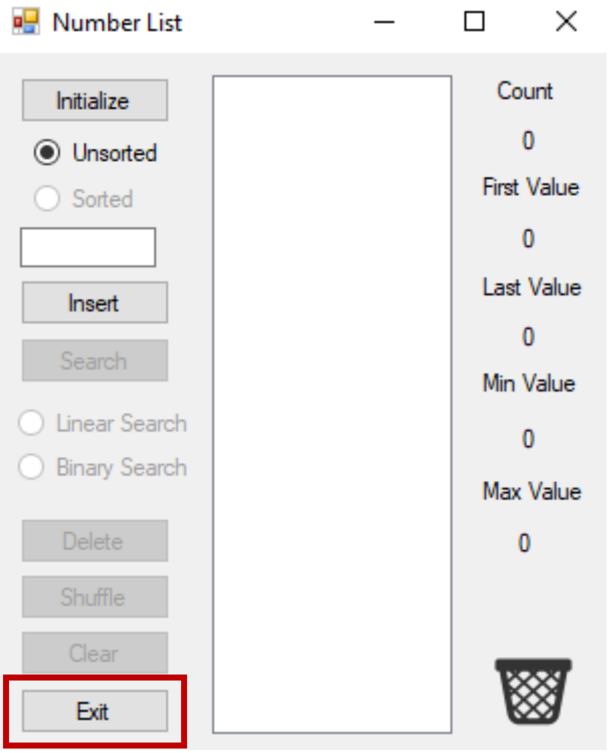
Please enter a numeric value!



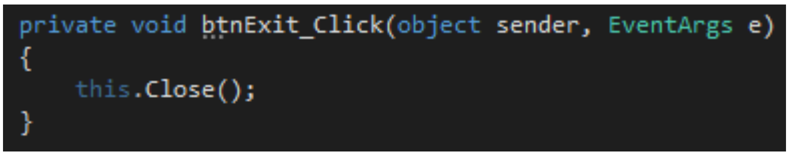
```
private void txtInsert_KeyPress(object sender, KeyPressEventArgs e)
{
    // prevents non-integer values from being entered e.g. string
    if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar))
    {
        e.Handled = true;
        MessageBox.Show("Please enter a numeric value!");
    }
}
```

Exit

Exit allows the user to exit the program.



'btnExit_Click' closes the form essentially exiting the program.



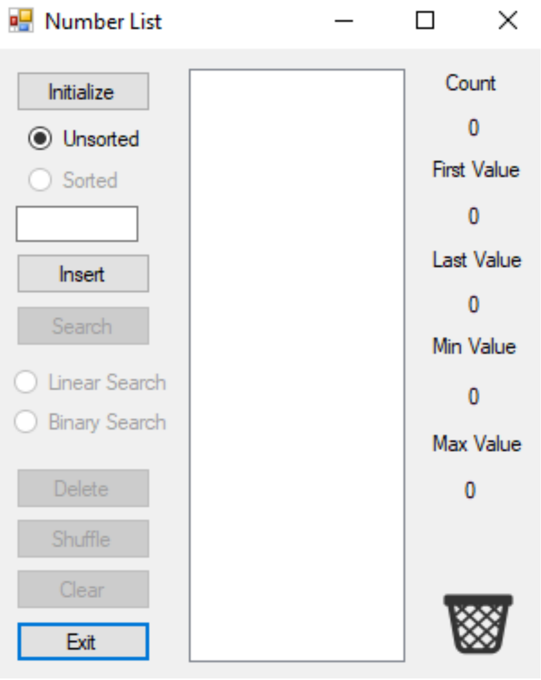
```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Test Plan

Test #1 – Buttons must be enabled and disabled as the state of the list changes

When the list initializes 30 values into the list, ‘Initialize’ and ‘Insert’ should disable and ‘Search’, ‘Delete’, ‘Shuffle’ and ‘Clear’ should enable.

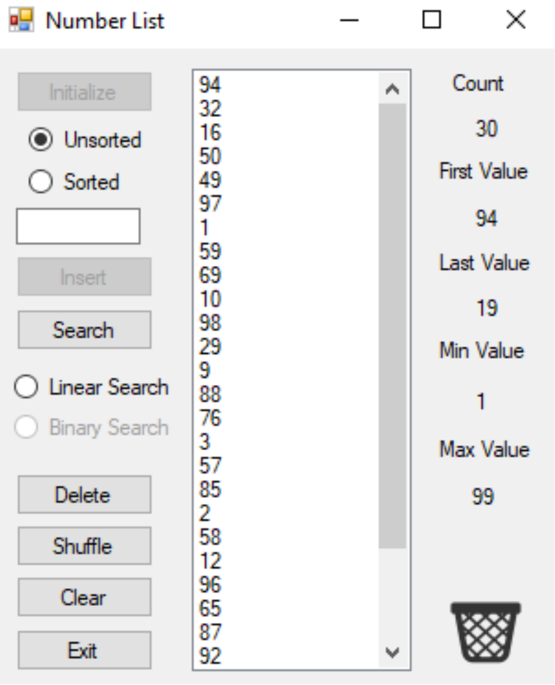
Before



● Unsorted
○ Sorted

Count
0
First Value
0
Last Value
0
Min Value
0
Max Value
0

After



● Unsorted
○ Sorted

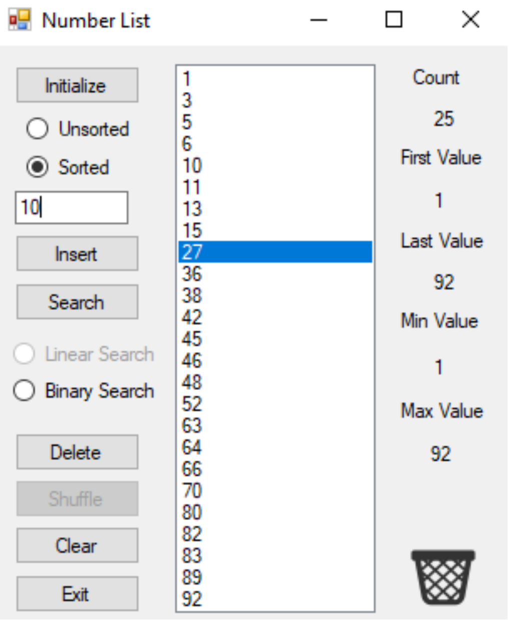
```
94
32
16
50
49
97
1
59
69
10
98
29
9
88
76
3
57
85
2
58
12
96
65
87
92
```

Count
30
First Value
94
Last Value
19
Min Value
1
Max Value
99

Test #2 – Attempting to insert a duplicate value

When attempting to insert a duplicate value to the list should display a messagebox stating "Error! Duplicate value!".

Before



○ Unsorted ● Sorted

10

1
3
5
6
10
11
13
15
27
36
38
42
45
46
48
52
63
64
66
70
80
82
83
89
92

Count 25

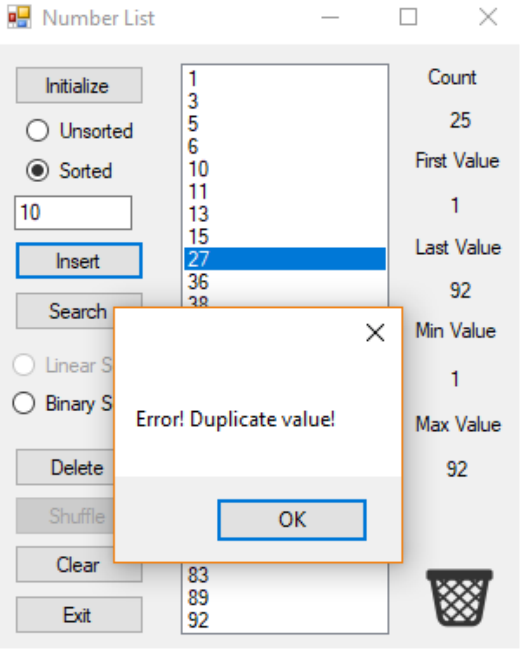
First Value 1

Last Value 92

Min Value 1

Max Value 92

After



○ Unsorted ● Sorted

10

1
3
5
6
10
11
13
15
27
36
38
83
89
92

Error! Duplicate value!

Count 25

First Value 1

Last Value 92

Min Value 1

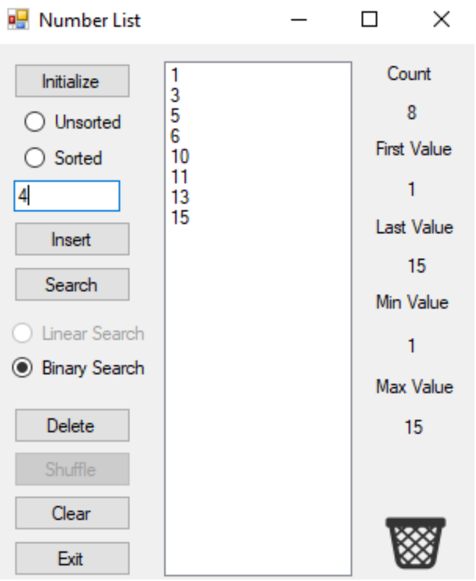
Max Value 92

Test #3 – Sorted List Searching

Searching for a non-existent number

Searching for a non-existent number should give an error message stating that “The value specified cannot be found”.

Before



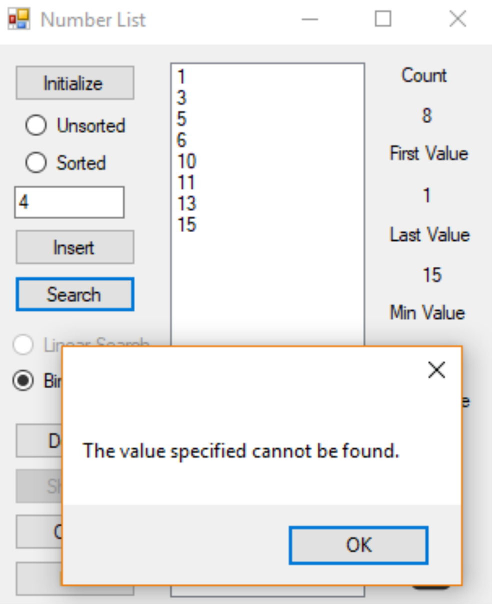
4

○ Linear Search
● Binary Search

1
3
5
6
10
11
13
15

Count
8
First Value
1
Last Value
15
Min Value
1
Max Value
15

After



4

○ Linear Search
● Binary Search

1
3
5
6
10
11
13
15

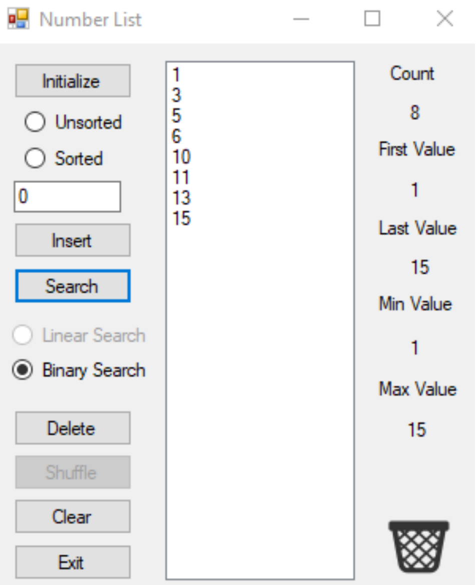
Count
8
First Value
1
Last Value
15
Min Value

The value specified cannot be found.

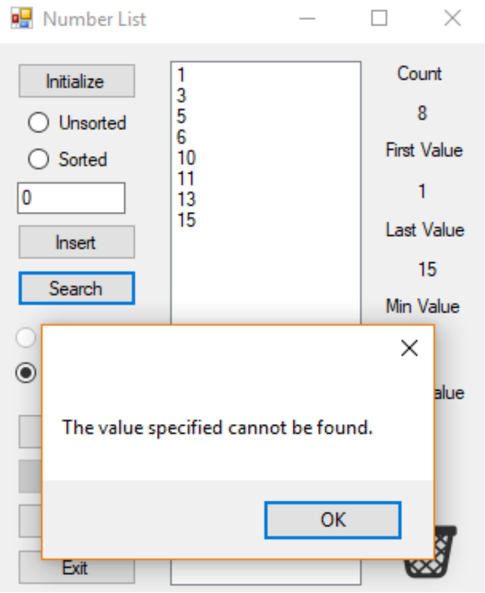
Searching for a non-existent number lower than the first value

Searching for a non-existent number lower than the first value should still give an error message stating that “The value specified cannot be found”.

Before



After

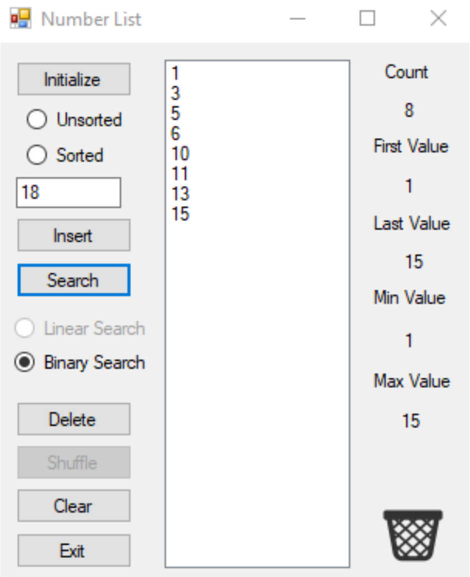


The value specified cannot be found.

Searching for a non-existent number greater than the last value

Searching for a non-existent number greater than the last value should still give an error message stating that "The value specified cannot be found".

Before

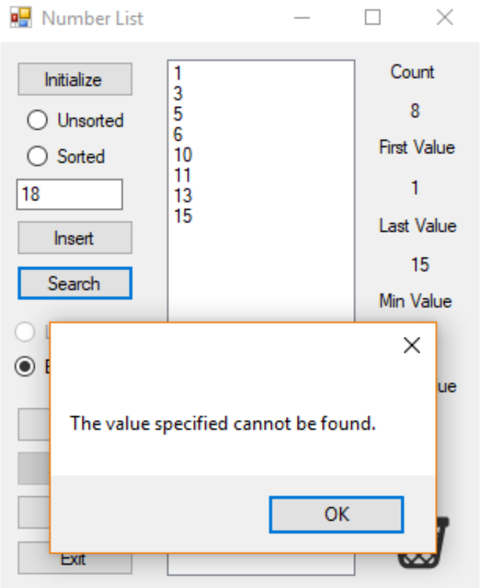


○ Unsorted
○ Sorted
18
○ Linear Search
● Binary Search

1
3
5
6
10
11
13
15

Count
8
First Value
1
Last Value
15
Min Value
1
Max Value
15

After



○ Unsorted
○ Sorted
18

1
3
5
6
10
11
13
15

Count
8
First Value
1
Last Value
15
Min Value

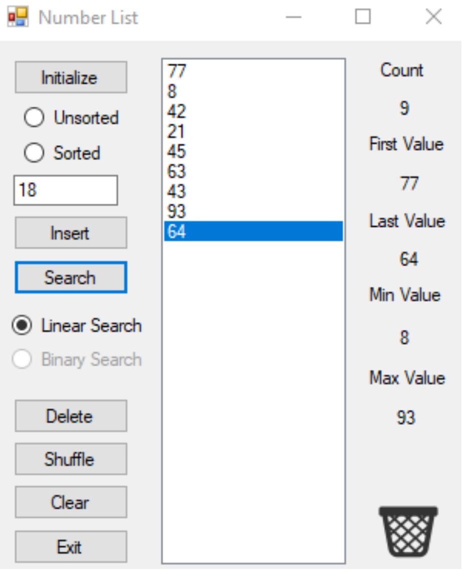
The value specified cannot be found.

Test #4 – Unsorted Search

Searching for a non-existent number

Searching for a non-existent number should still give an error message stating that "The value specified cannot be found".

Before



○ Unsorted ○ Sorted

18

77
8
42
21
45
63
43
93
64

● Linear Search ○ Binary Search

Count 9

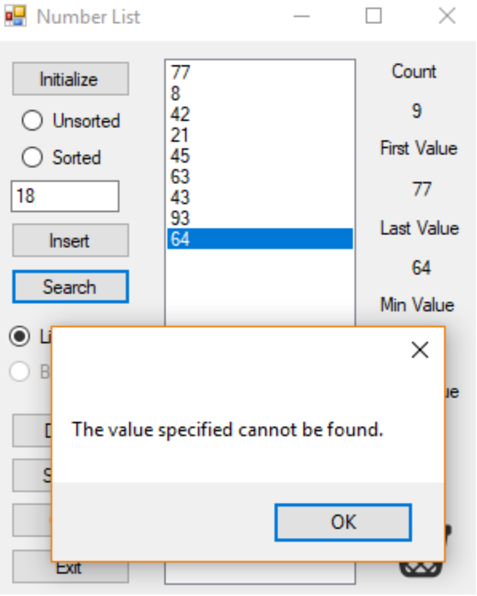
First Value 77

Last Value 64

Min Value 8

Max Value 93

After



18

77
8
42
21
45
63
43
93
64

Count 9

First Value 77

Last Value 64

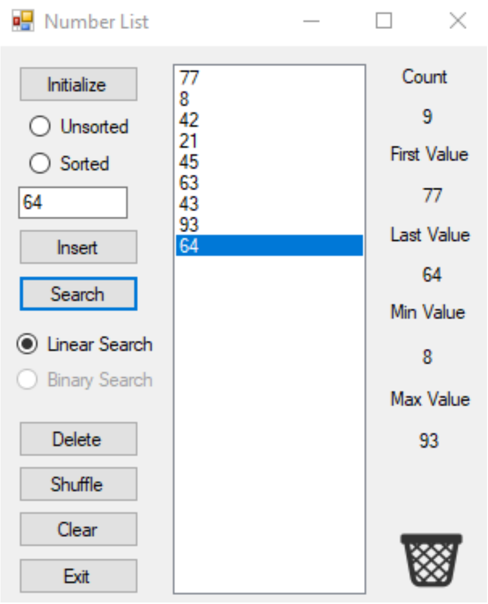
Min Value

The value specified cannot be found.

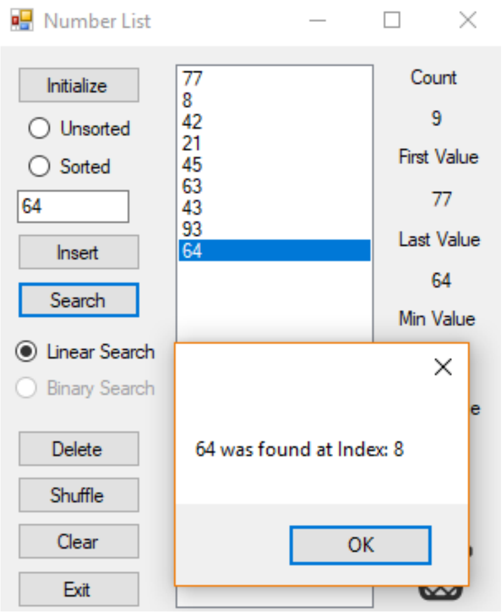
Searching for the last number

The value specified should still be found with a messagebox showing stats on where that value was found.

Before



After

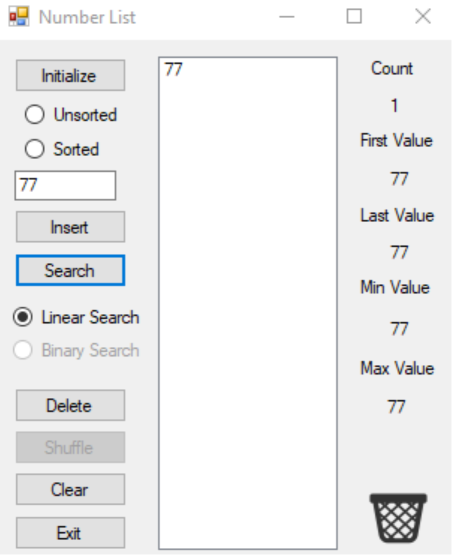


64 was found at Index: 8

Test Plan #5 – Searching when only one entry exists

When searching a list with only one item, it should still find the value specified.

Before



○ Unsorted

○ Sorted

77

● Linear Search

○ Binary Search

77

Count 1

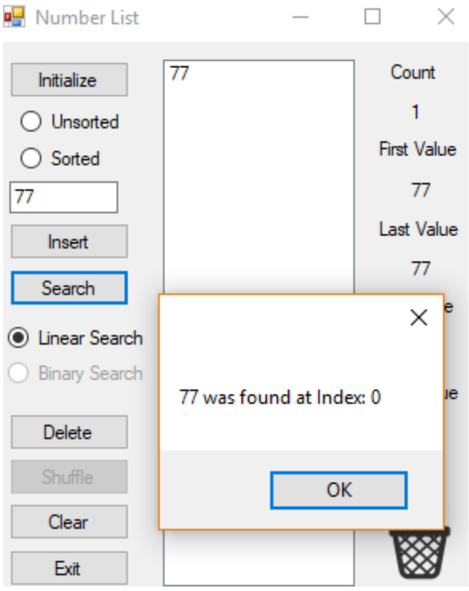
First Value 77

Last Value 77

Min Value 77

Max Value 77

After



○ Unsorted

○ Sorted

77

● Linear Search

○ Binary Search

77

Count 1

First Value 77

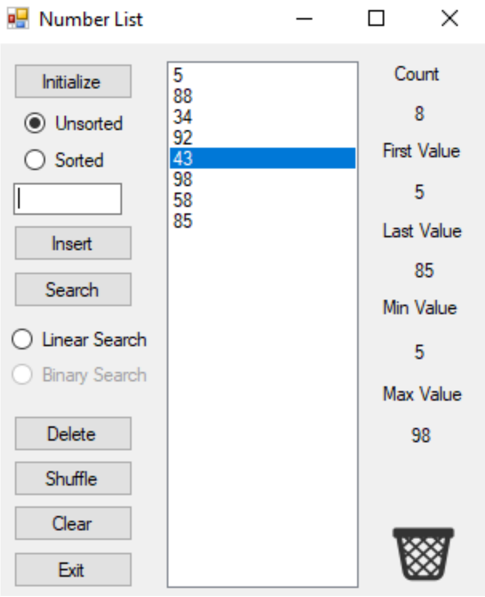
Last Value 77

77 was found at Index: 0

Test Plan #6 – Initializing the List

The list should be filled by adding values until the max count of 30 is reached regardless of whether there are items already within the list e.g. 14 items already in the list so initialize adds 16 only.

Before



● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

```
5
88
34
92
43
98
58
85
```

Count

8

First Value

5

Last Value

85

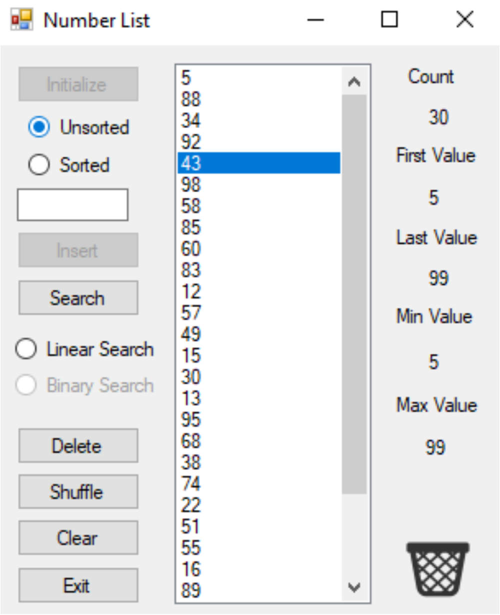
Min Value

5

Max Value

98

After



● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

```
5
88
34
92
43
98
58
85
60
83
12
57
49
15
30
13
95
68
38
74
22
51
55
16
89
```

Count

30

First Value

5

Last Value

99

Min Value

5

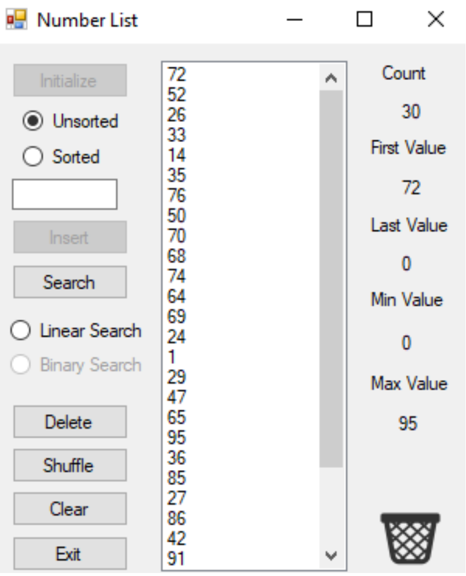
Max Value

99

Test Plan #7 – Sorting the List & Sort Functionality

Checking the ‘Sorted’ radio button should sort the list and allow the user to use sorted functionality such as binary search or an ordered insert

Before

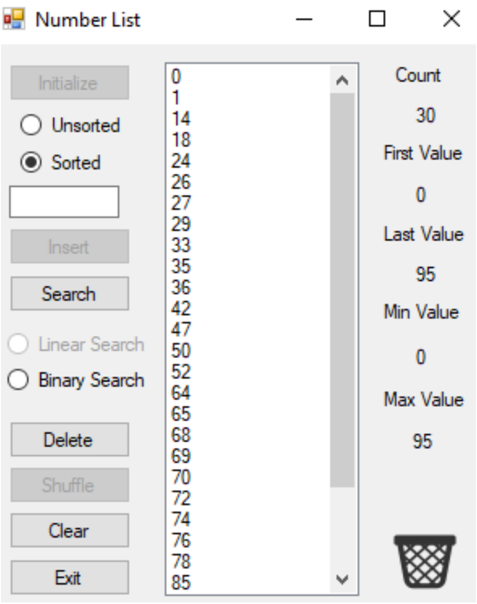


● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

72
52
26
33
14
35
76
50
70
68
74
64
69
24
1
29
47
65
95
36
85
27
86
42
91

Count
30
First Value
72
Last Value
0
Min Value
0
Max Value
95

After



○ Unsorted
● Sorted
○ Linear Search
○ Binary Search

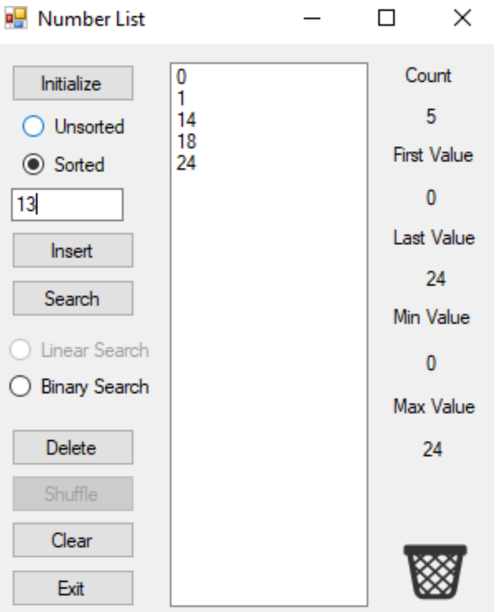
0
1
14
18
24
26
27
29
33
35
36
42
47
50
52
64
65
68
69
70
72
74
76
78
85

Count
30
First Value
0
Last Value
95
Min Value
0
Max Value
95

Test Plan #8 – Ordered Insert

Using insert when sorted is checked, it should insert the value in a sorted manner.

Before



Number List

○ Unsorted

● Sorted

13

○ Linear Search

○ Binary Search

```
0
1
14
18
24
```

Count

5

First Value

0

Last Value

24

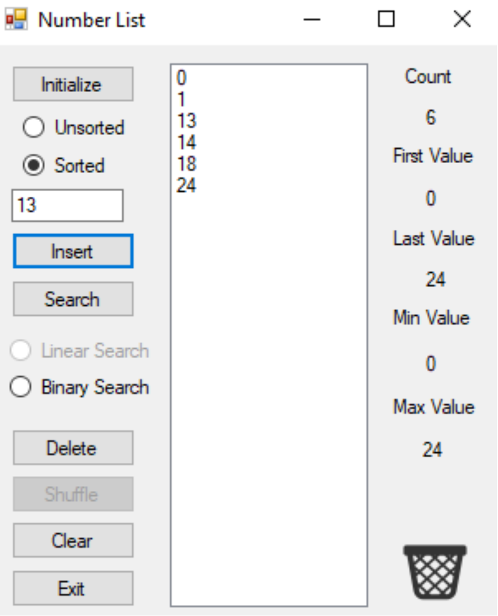
Min Value

0

Max Value

24

After



Number List

○ Unsorted

● Sorted

13

○ Linear Search

○ Binary Search

```
0
1
13
14
18
24
```

Count

6

First Value

0

Last Value

24

Min Value

0

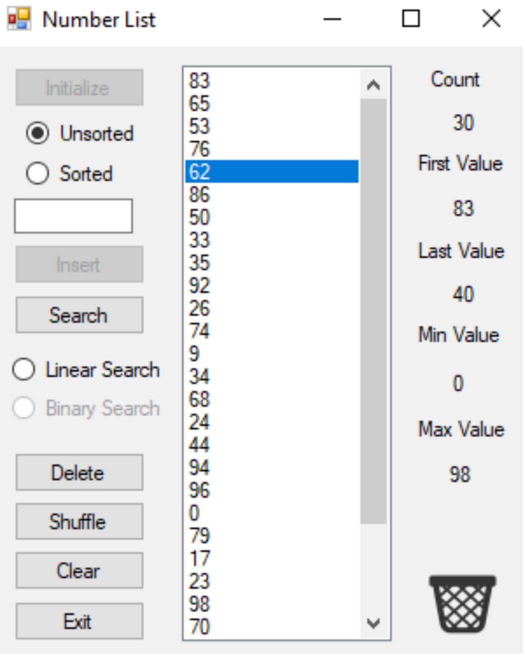
Max Value

24

Test Plan #9 – Delete

Selecting a value from the list and pressing delete should remove the selected value from the list.

Before

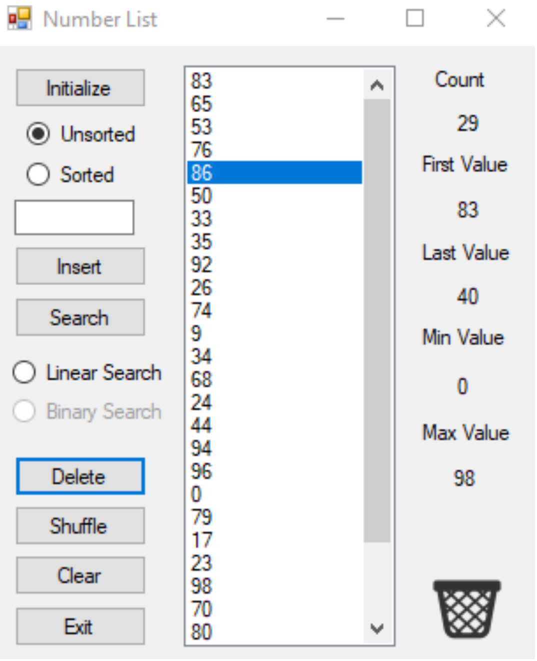


● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

```
83
65
53
76
62
86
50
33
35
92
26
74
9
34
68
24
44
94
96
0
79
17
23
98
70
```

Count
30
First Value
83
Last Value
40
Min Value
0
Max Value
98

After



● Unsorted
○ Sorted
○ Linear Search
○ Binary Search

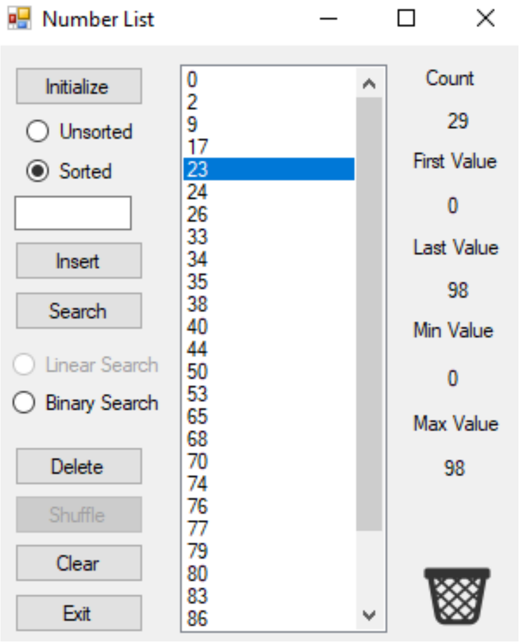
```
83
65
53
76
86
50
33
35
92
26
74
9
34
68
24
44
94
96
0
79
17
23
98
70
80
```

Count
29
First Value
83
Last Value
40
Min Value
0
Max Value
98

Test Plan #10 – Shuffling the List

The list should be shuffled when pressing ‘Shuffle’.

Before



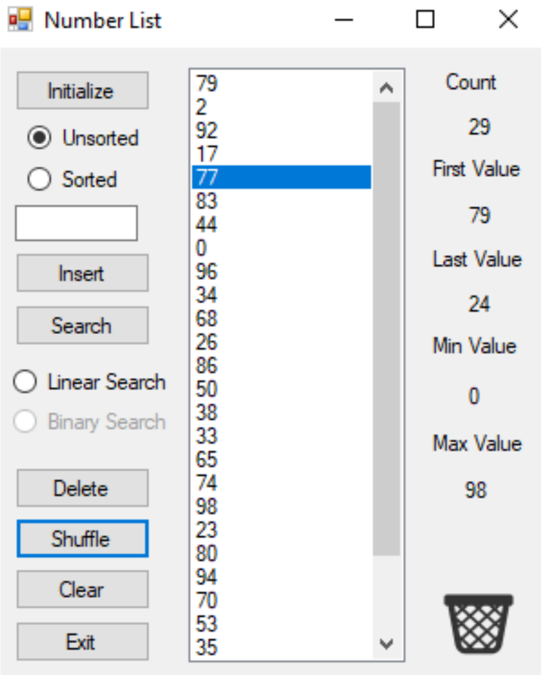
○ Unsorted
● Sorted

○ Linear Search
○ Binary Search

```
0
2
9
17
23
24
26
33
34
35
38
40
44
50
53
65
68
70
74
76
77
79
80
83
86
```

Count
29
First Value
0
Last Value
98
Min Value
0
Max Value
98

After



● Unsorted
○ Sorted

○ Linear Search
○ Binary Search

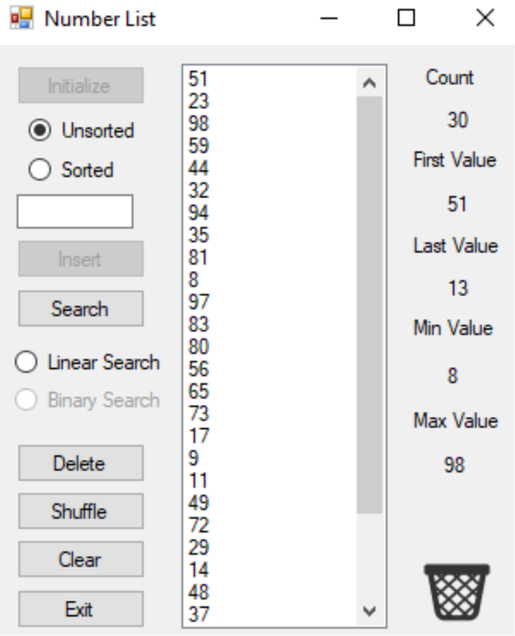
```
79
2
92
17
77
83
44
0
96
34
68
26
86
50
38
33
65
74
98
23
80
94
70
53
35
```

Count
29
First Value
79
Last Value
24
Min Value
0
Max Value
98

Test Plan #11 – Clear

The list should be able cleared (empty list) when ‘Clear’ is used.

Before

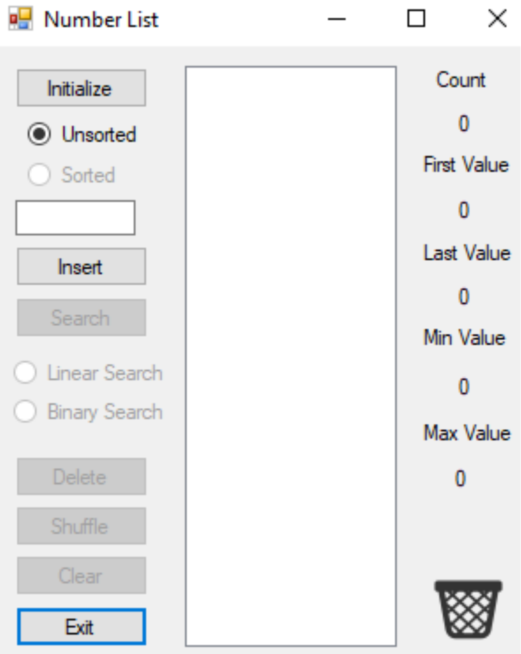


● Unsorted
○ Sorted
○ Linear Search

51
23
98
59
44
32
94
35
81
8
97
83
80
56
65
73
17
9
11
49
72
29
14
48
37

Count
30
First Value
51
Last Value
13
Min Value
8
Max Value
98

After



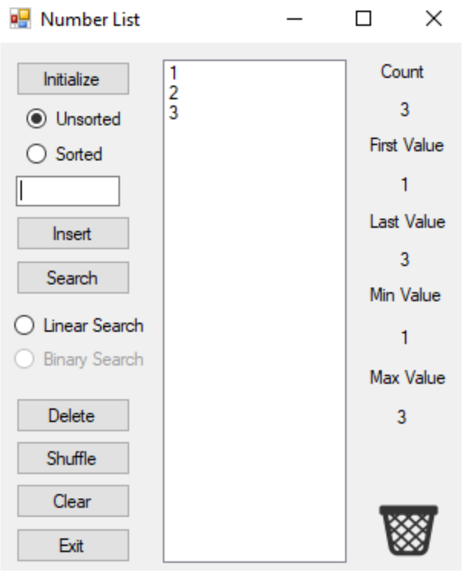
● Unsorted
○ Sorted

Count
0
First Value
0
Last Value
0
Min Value
0
Max Value
0

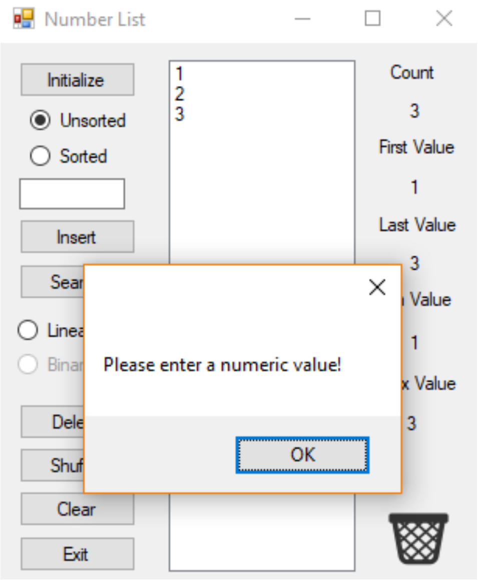
Test Plan #12 – Attempting to input non-integer values to the textbox

The textbox shouldn't allow any non-integer values on keypress to be inputted by the user.

Before



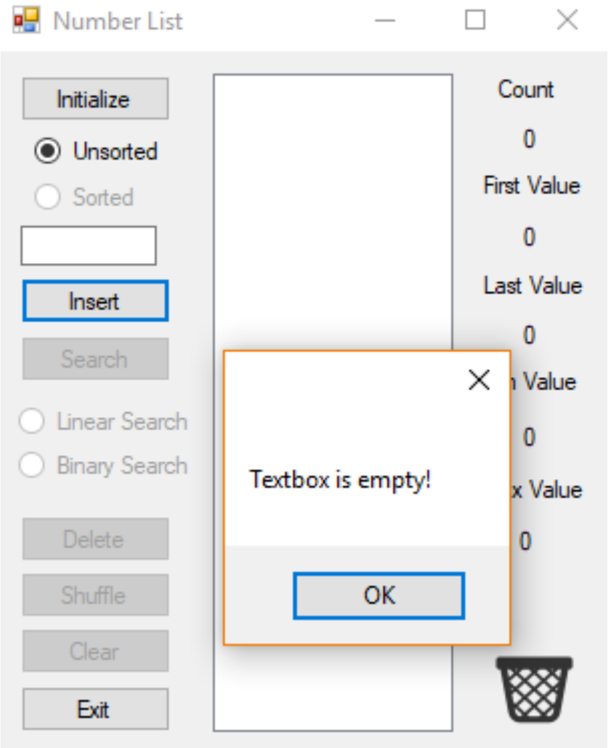
After



Please enter a numeric value!

Test Plan #13 – Attempting to insert null/whitespace

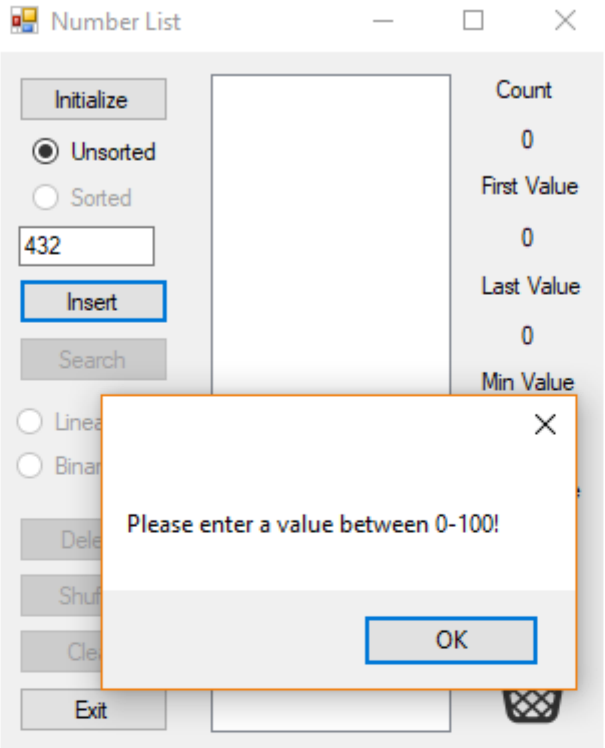
The user shouldn't be allowed to input an empty textbox which only contains whitespace.



Textbox is empty!

Test Plan #14 – Inputting values out of range

The user shouldn't be allowed to insert any values outside the range of 0 – 100.



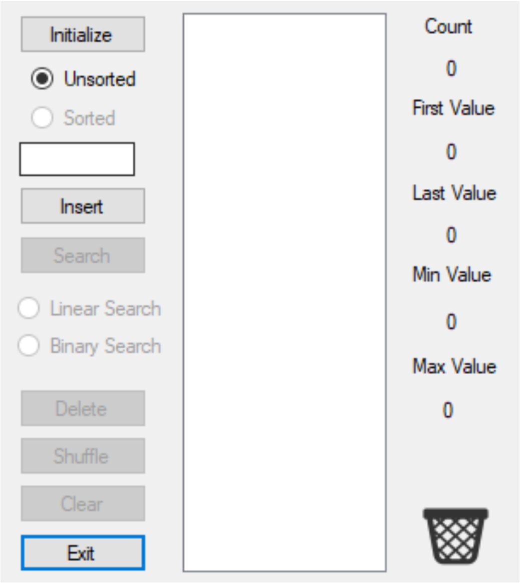
432

Please enter a value between 0-100!

Test Plan #15 – Status Updates

The program should update values anytime the list changes.

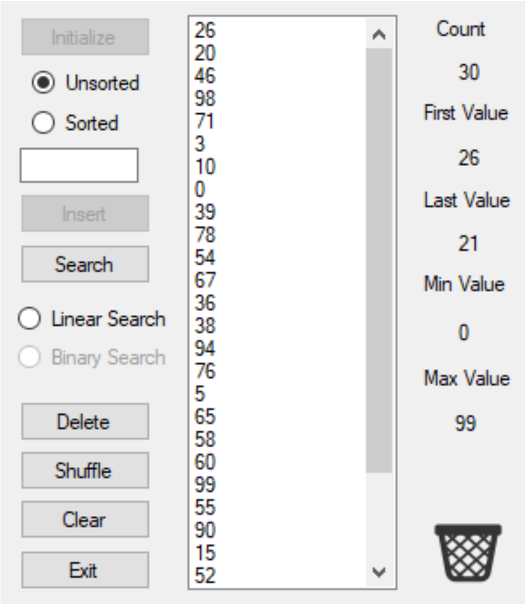
Before



● Unsorted
○ Sorted

Count
0
First Value
0
Last Value
0
Min Value
0
Max Value
0

After



● Unsorted
○ Sorted

```
26
20
46
98
71
3
10
0
39
78
54
67
36
38
94
76
5
65
58
60
99
55
90
15
52
```

Count
30
First Value
26
Last Value
21
Min Value
0
Max Value
99