COSC 2306
Data Programming

Sort

Sorting

- **Sorting takes an unordered collection and makes it an ordered one**

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

Sorting

- Sort a list of words alphabetically or by length
- Sort a list of cities by population, area , GDP, zip code
- Sorting helps search
- Sorting can be expensive
  - Compare items (number of comparisons)
  - Exchange item (swap)
- Efficiency of sorting is critical

Bubble Sort

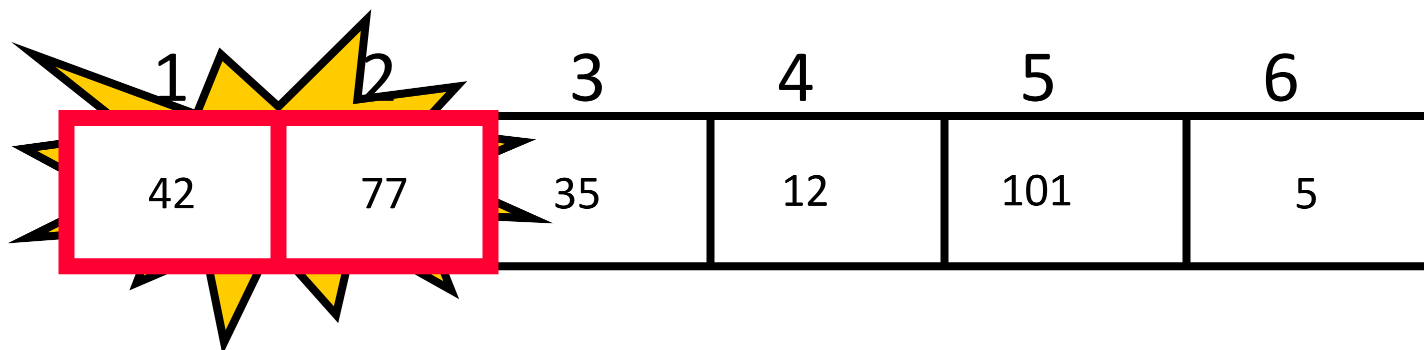
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  – Move from the front to the end
  – "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

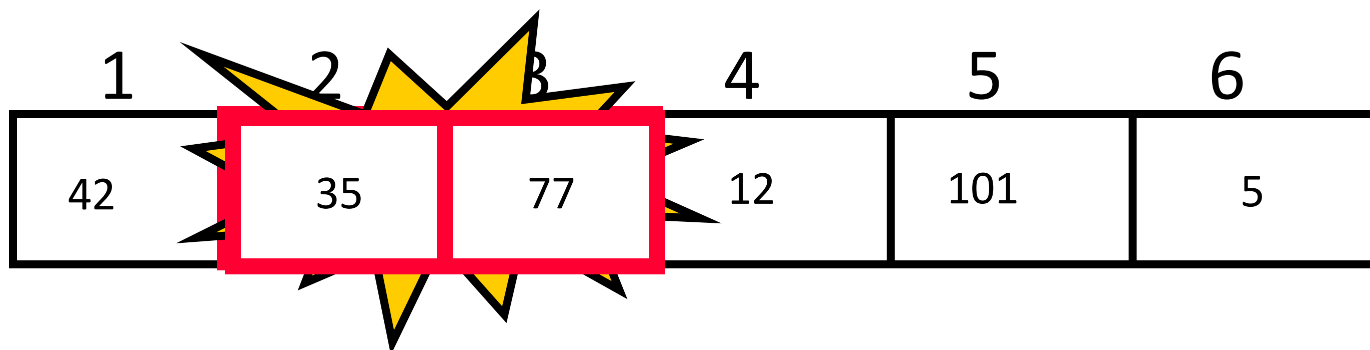
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



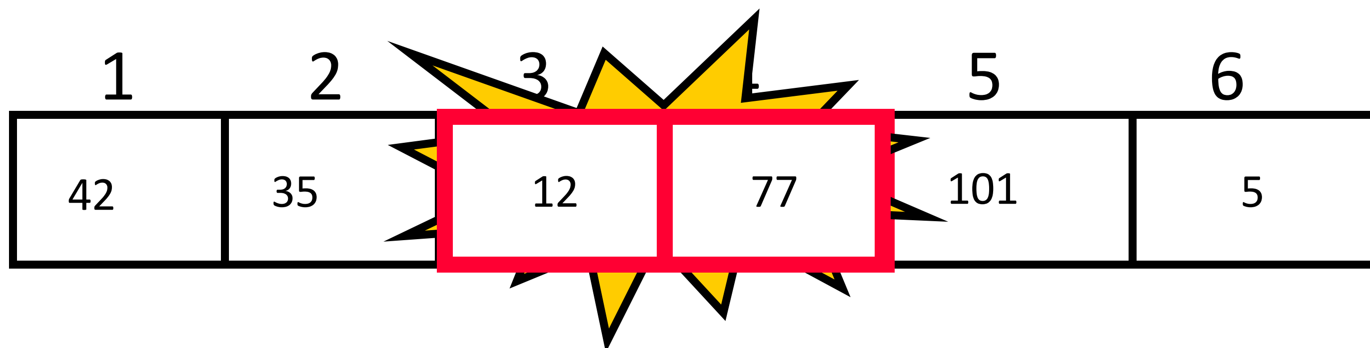
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



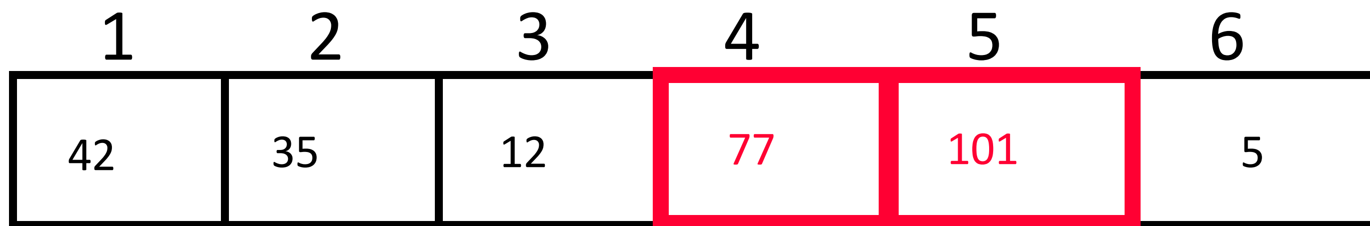
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



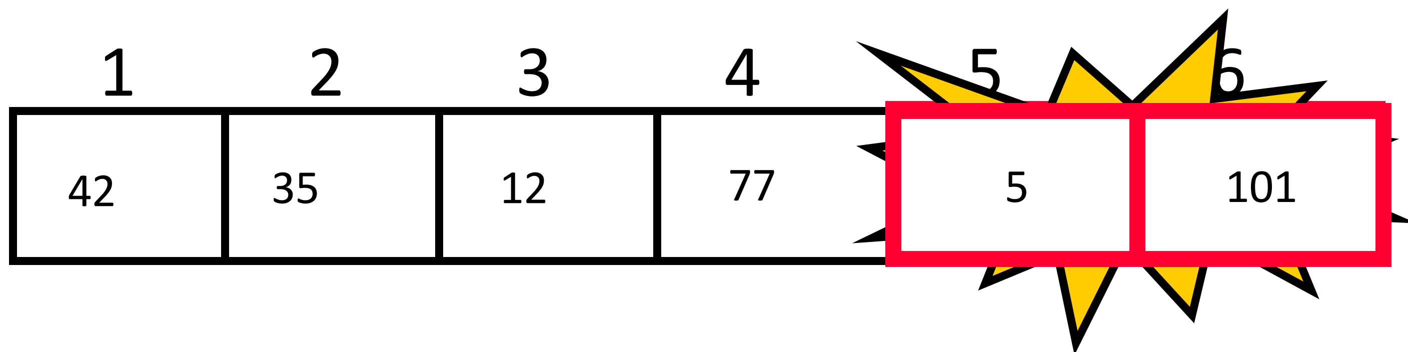
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



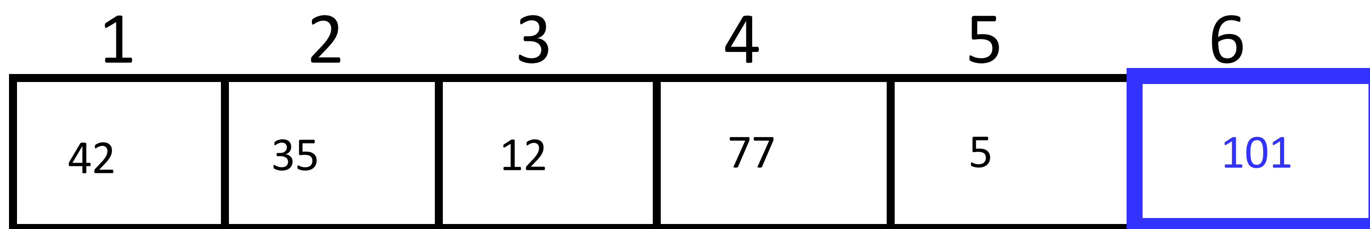
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



Largest value correctly placed

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

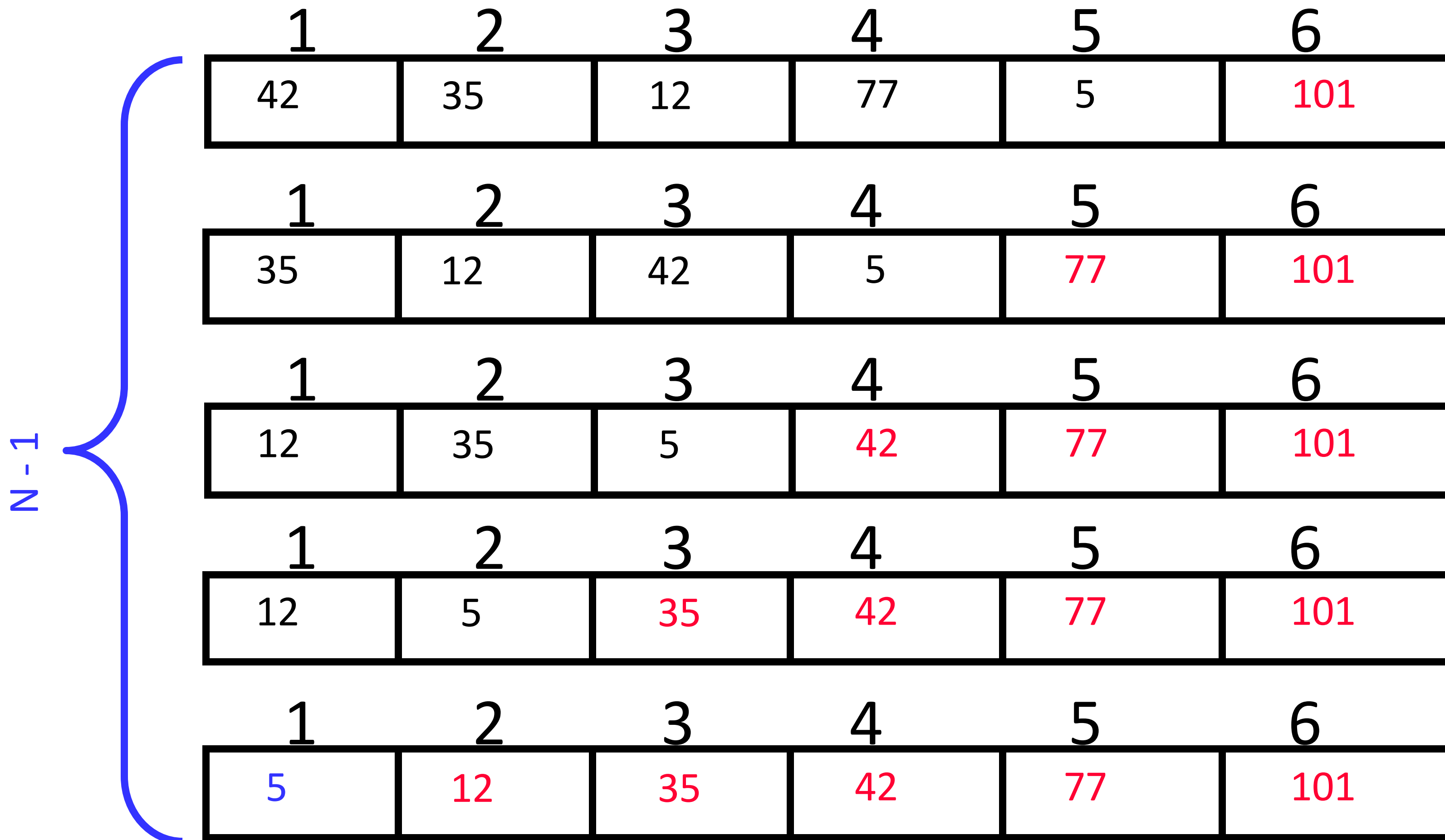
1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Repeat "Bubble Up" How Many Times?

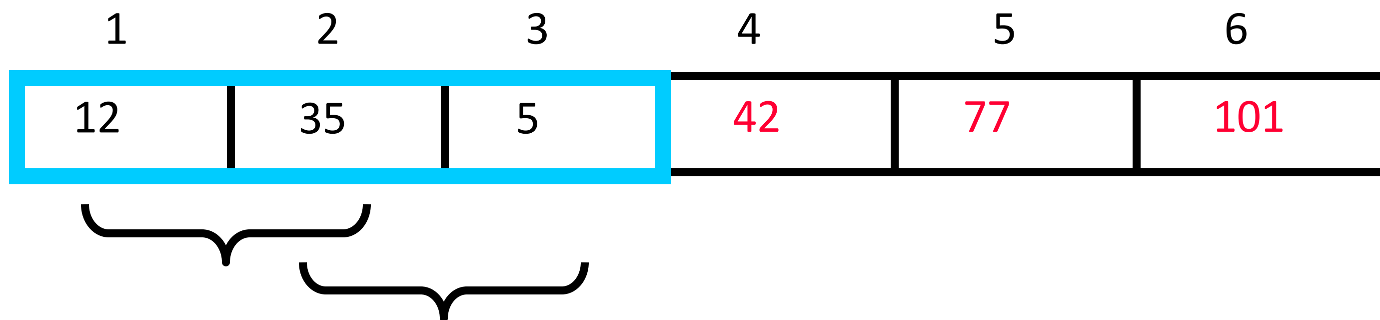
- If we have N elements
- And if each time we bubble an element, we place it in its correct location
- We repeat the "bubble up" process N – 1 times
- This guarantees we'll correctly place all N elements

"Bubbling" All the Elements



Number of Comparisons

- On the Nth "bubble up", we only need to do MAX-N comparisons
- For example:
  - This is the 4th "bubble up"
  - MAX is 6
  - We have 2 comparisons



Bubble up coding

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
$\dots$	$\dots$
$n-1$	1

Need a nested for loop structure

Need swap code: if a > b, swap a and b

```
if (a > b):
    temp = a
    a = b
    b = temp
```

```
if (a > b):
    a, b = b, a
```

Bubble up coding

```
def bubble_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(a_list)
print(a_list)
```

Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?

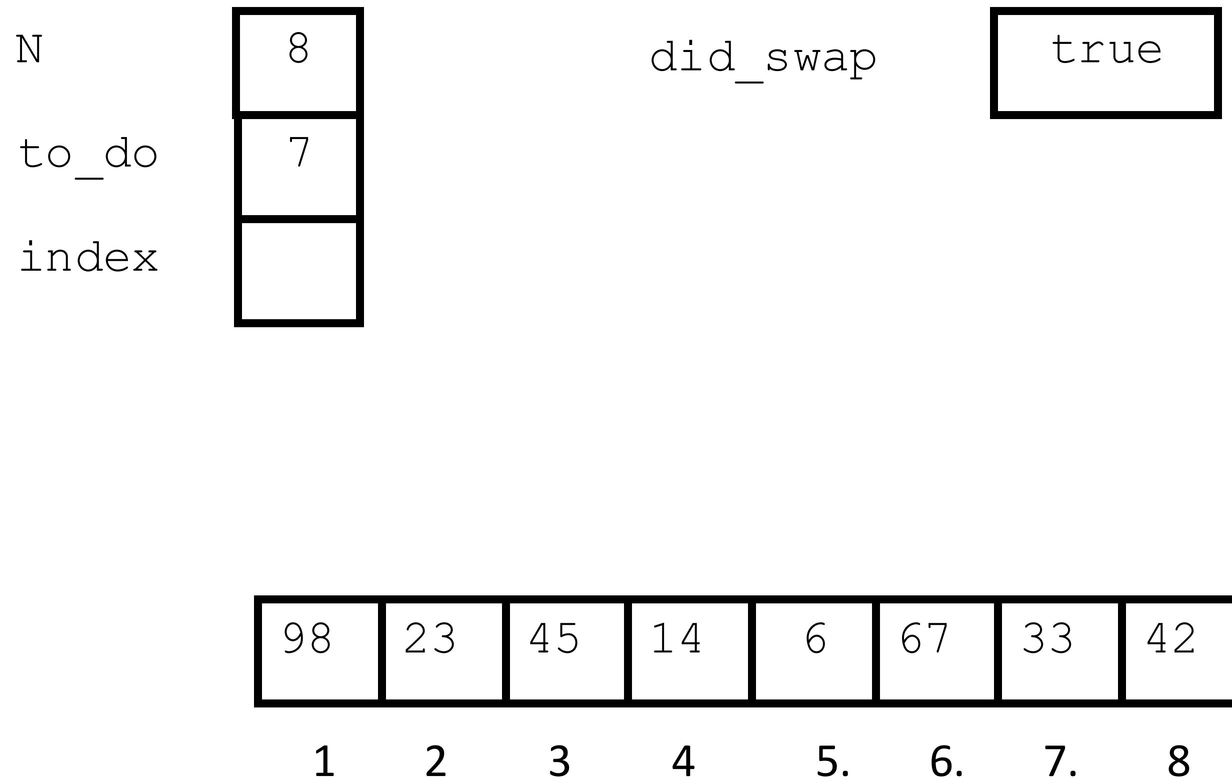
1	2	3	4	5	6
5	12	35	42	77	101

- We can detect this to "stop early"!

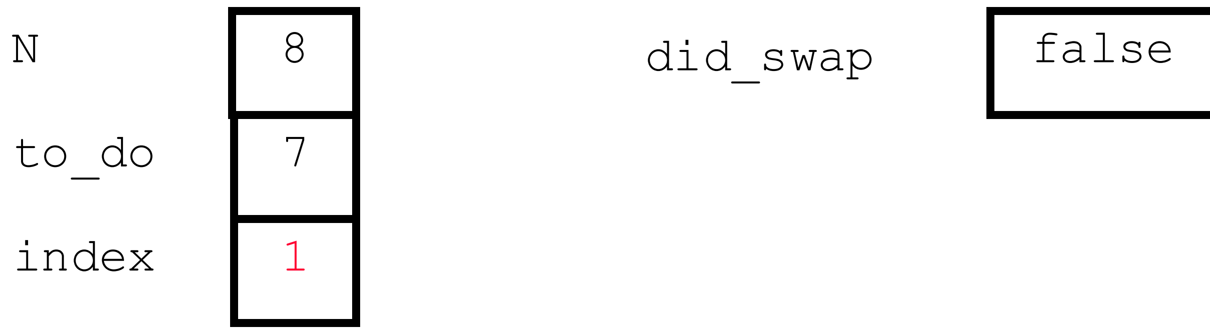
Using a Boolean “Flag”

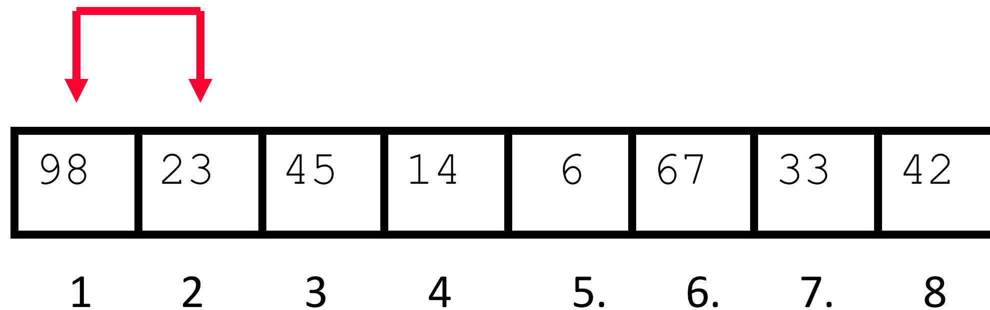
- We can use a boolean variable to determine if any swapping occurred during the “bubble up”
- If no swapping occurred, then we know that the collection is already sorted
- This boolean “flag” needs to be reset after each “bubble up”

An Animated Example

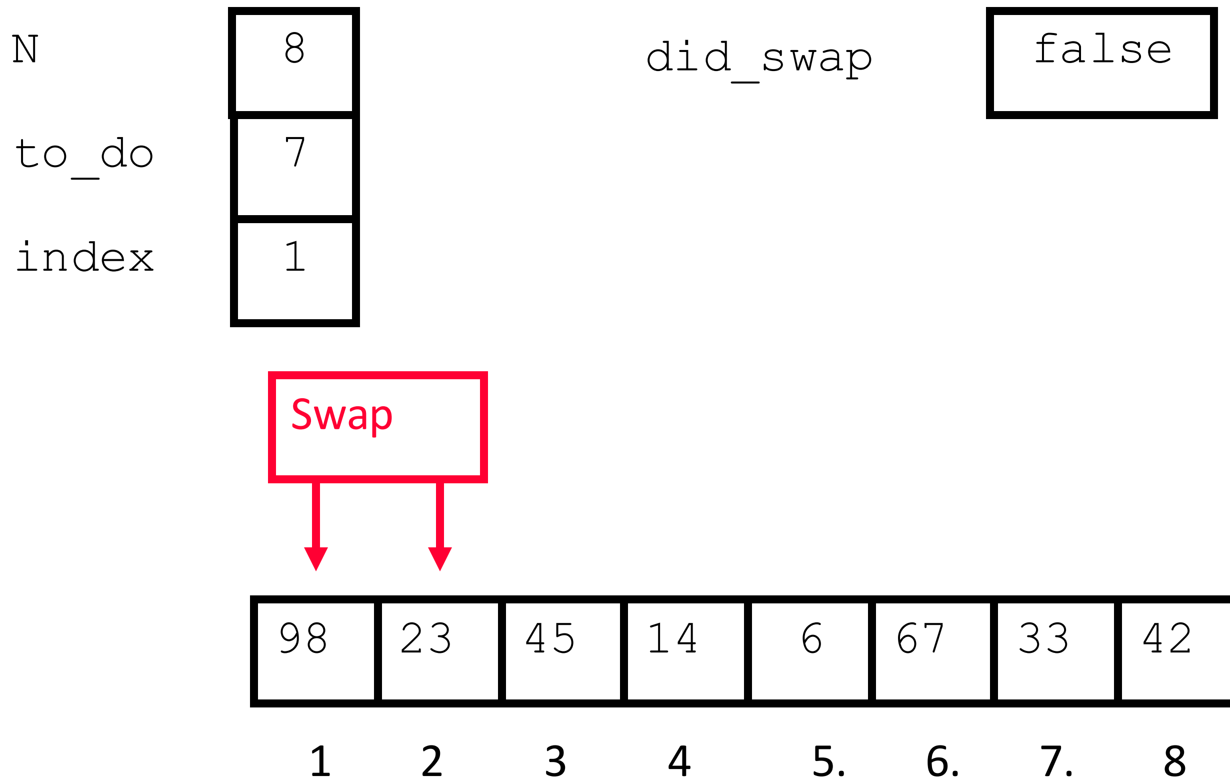


An Animated Example

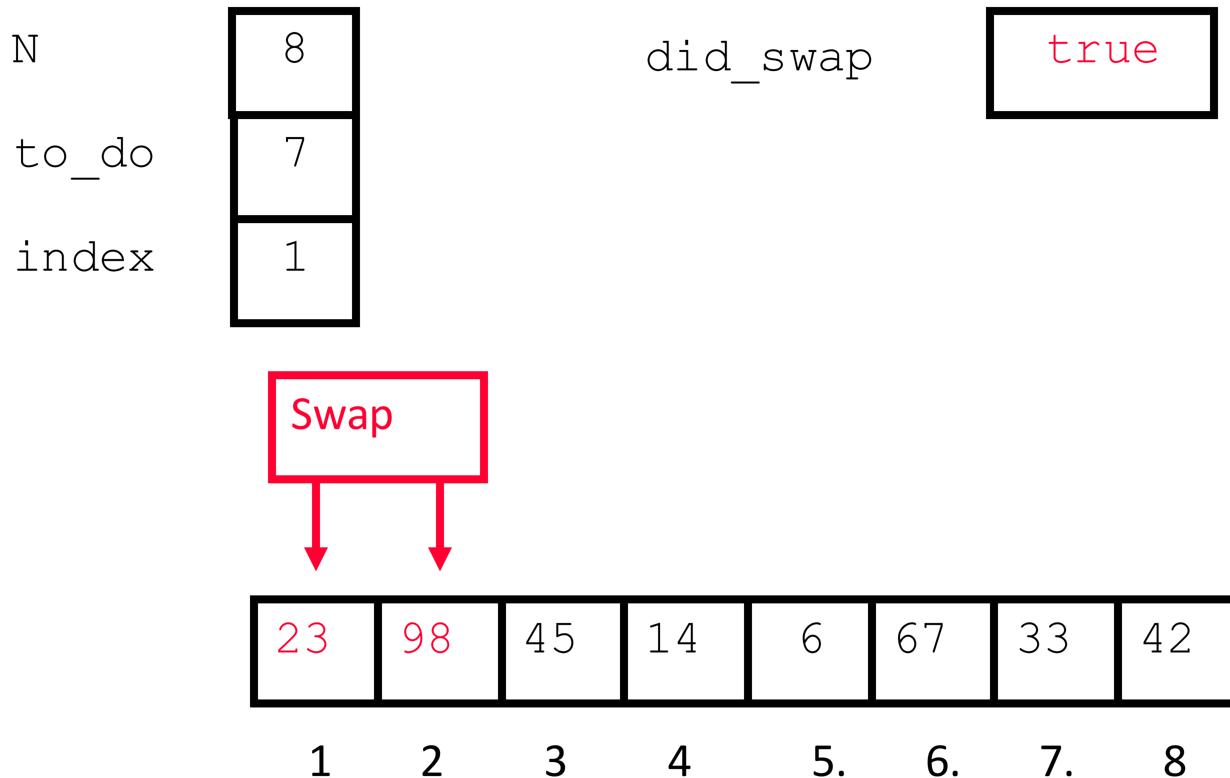




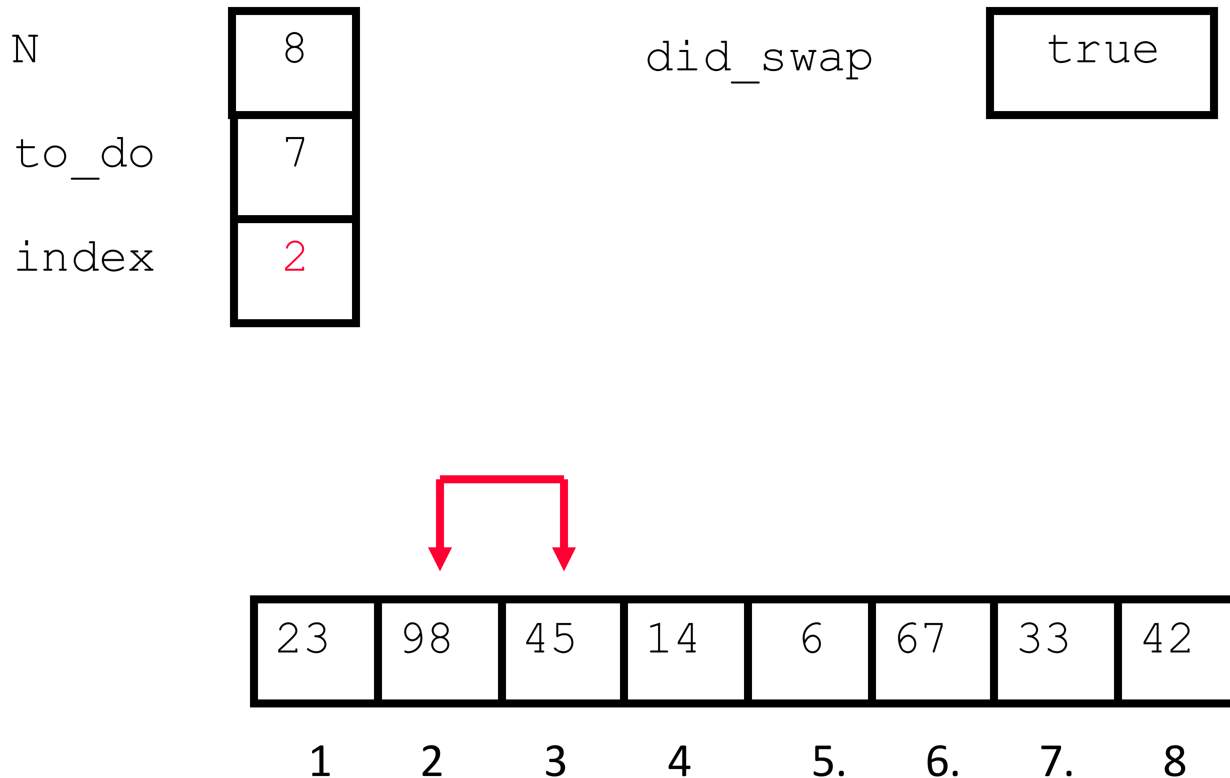
An Animated Example



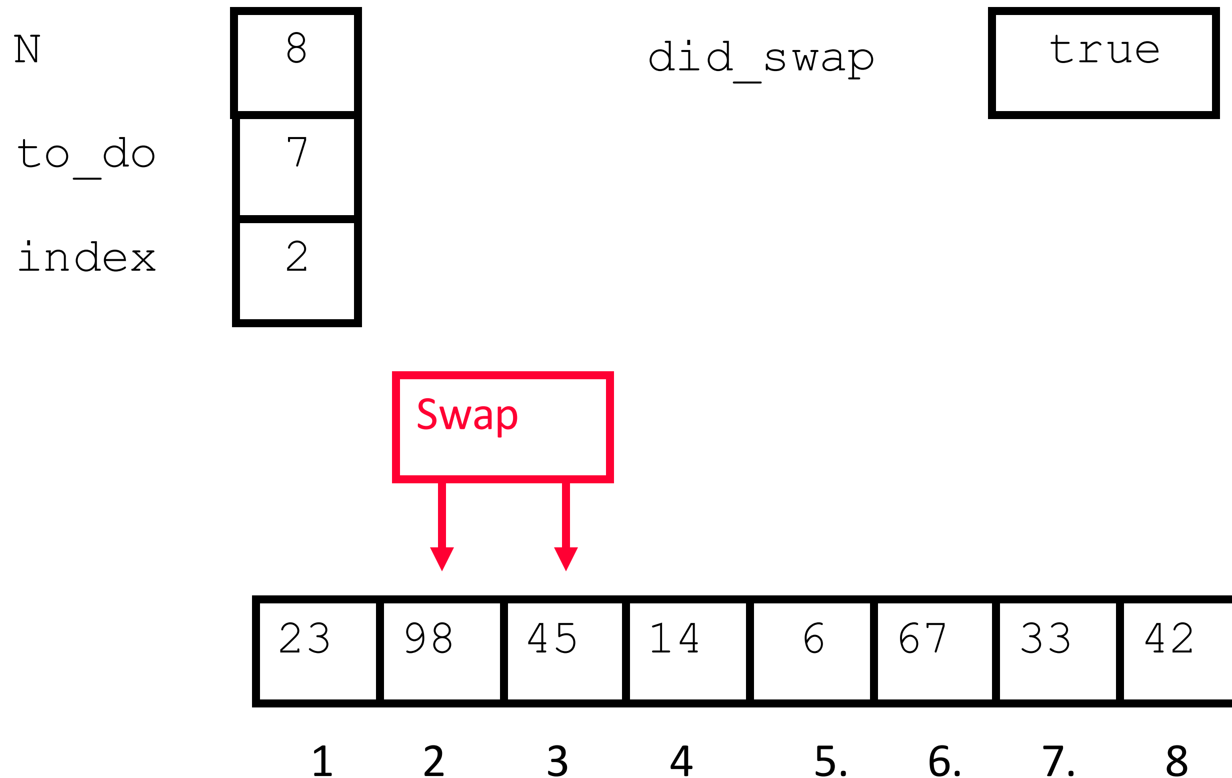
An Animated Example



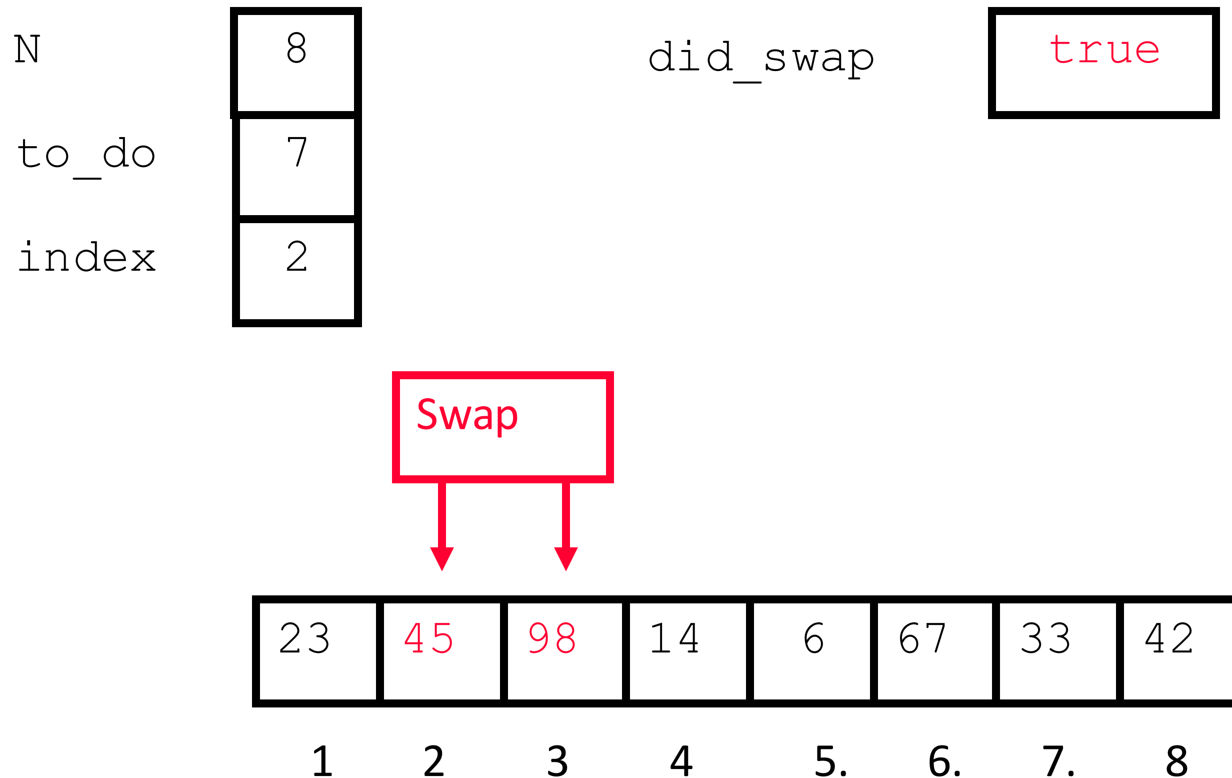
An Animated Example



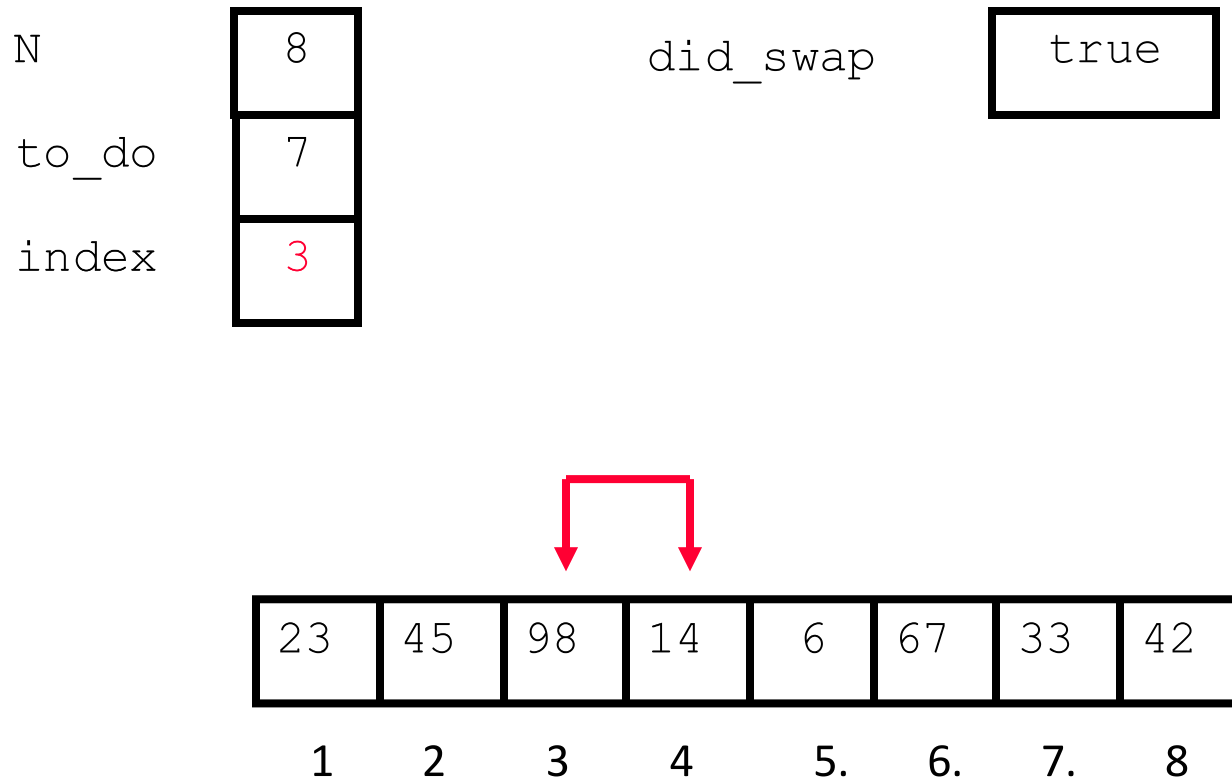
An Animated Example



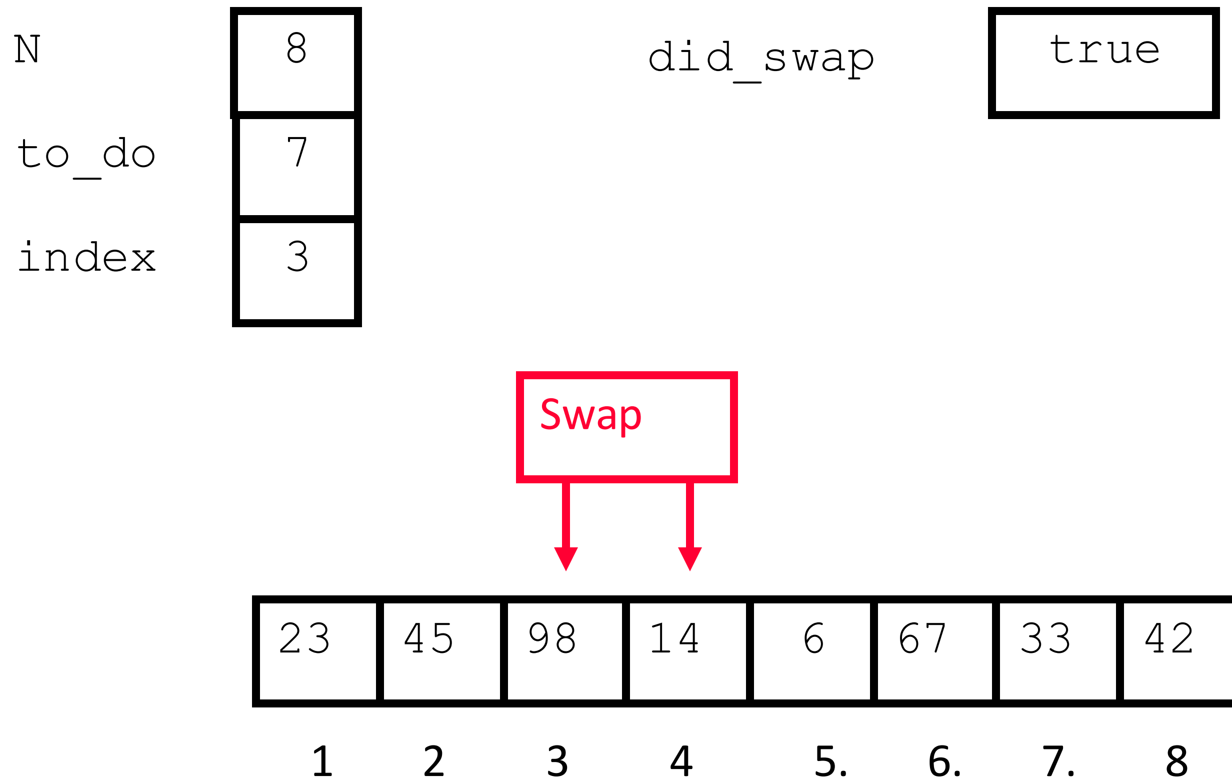
An Animated Example



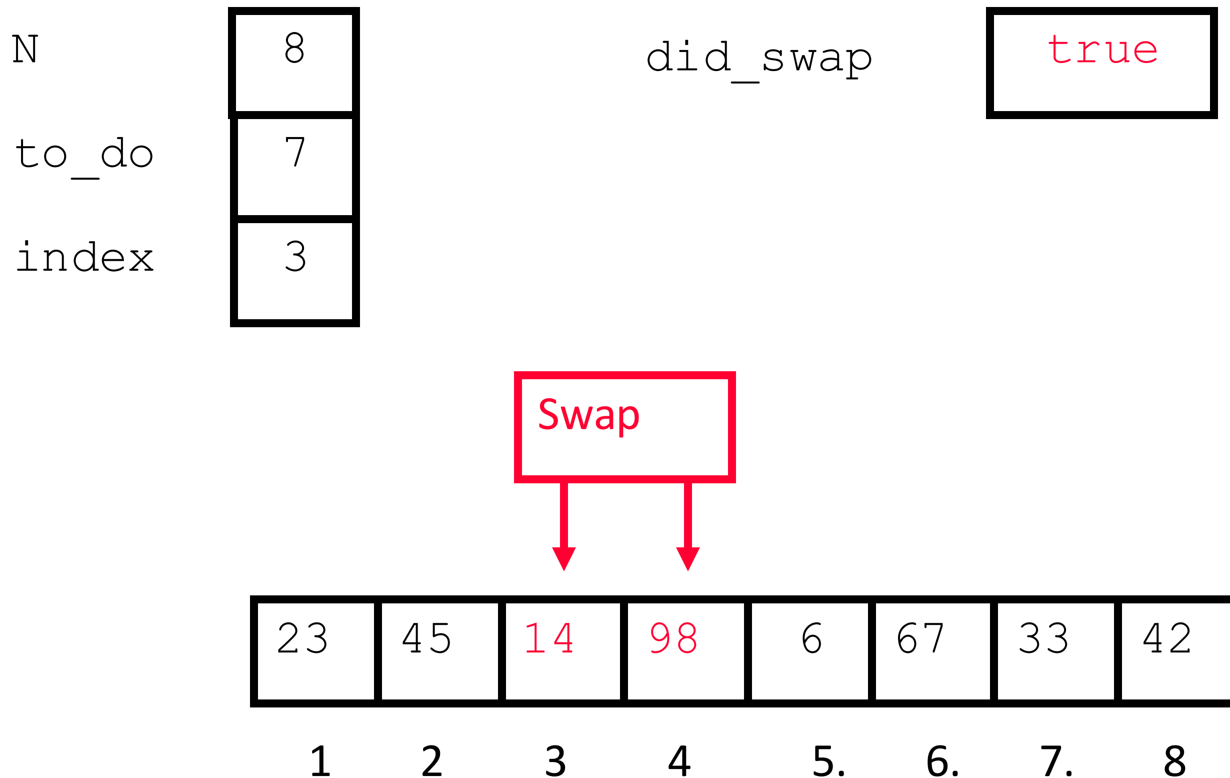
An Animated Example



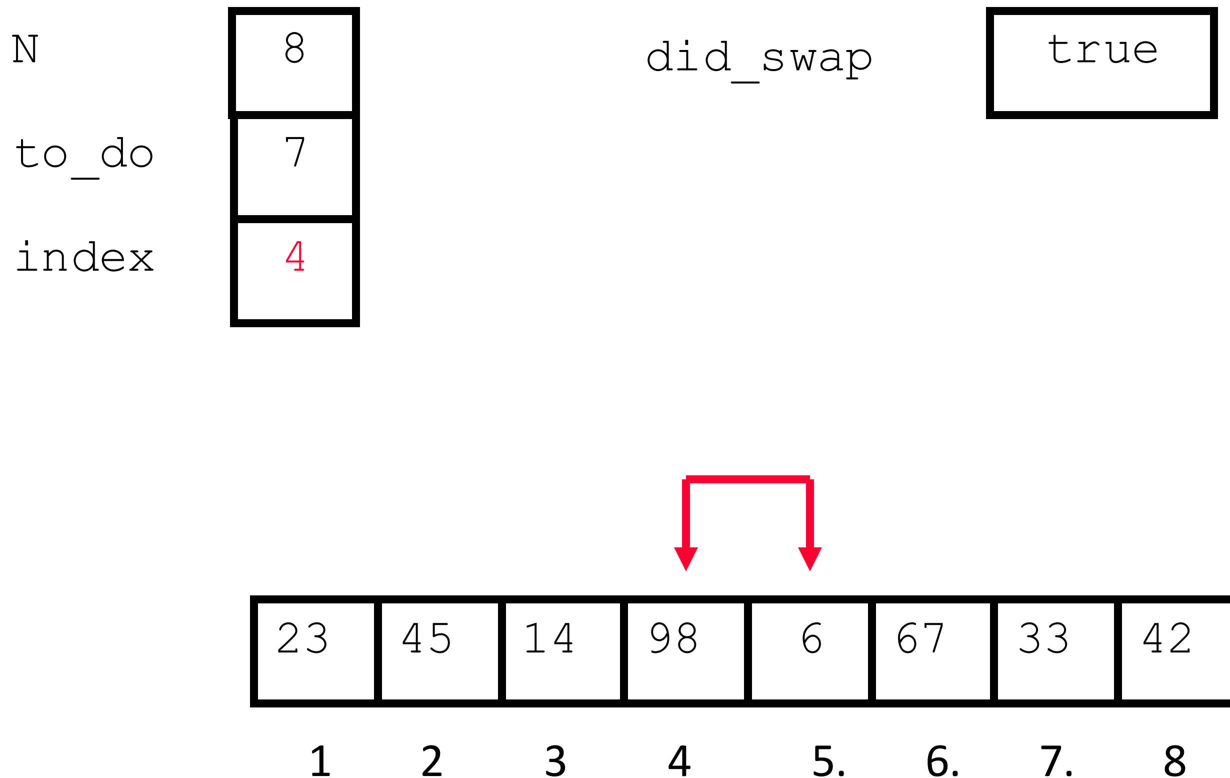
An Animated Example



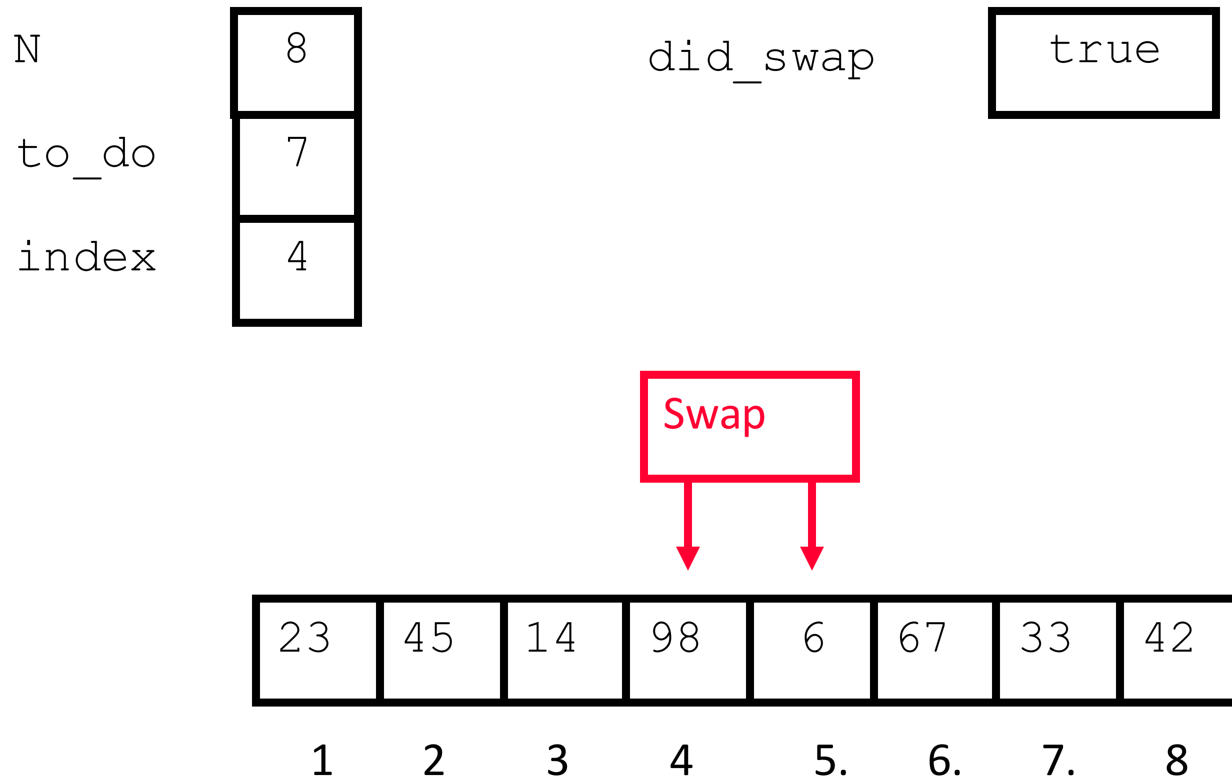
An Animated Example



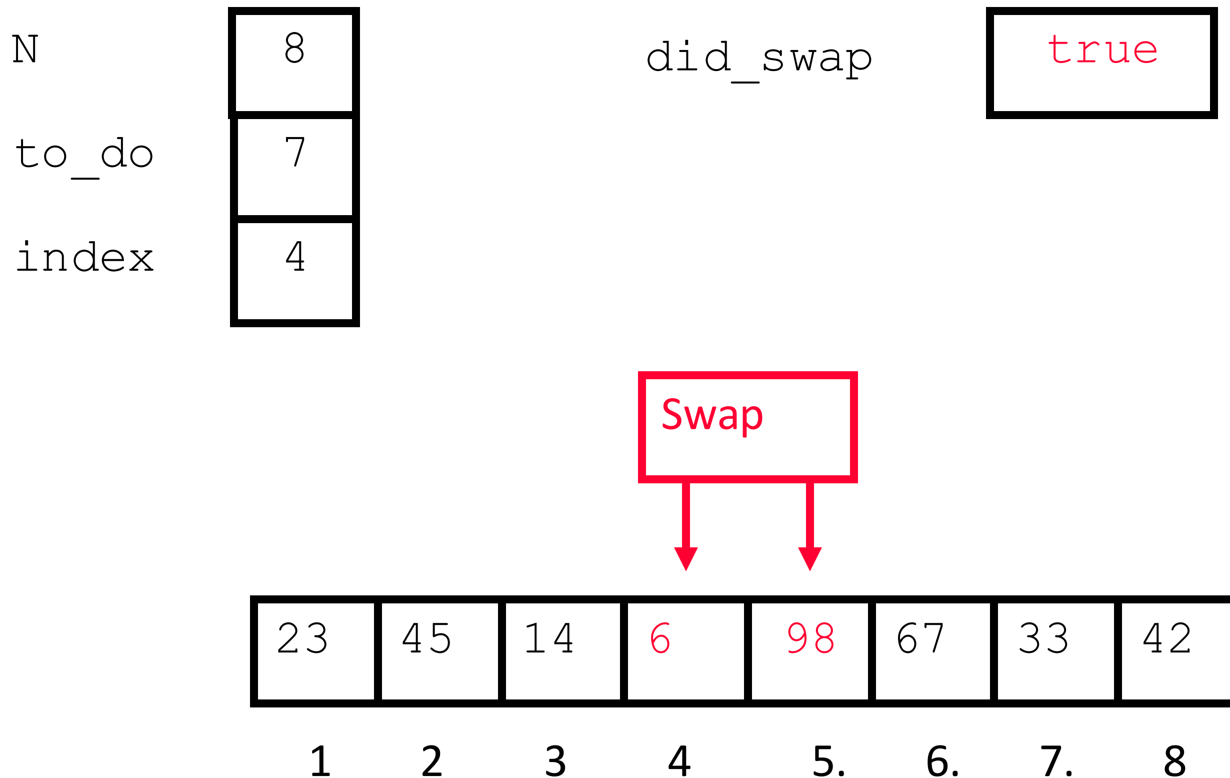
An Animated Example



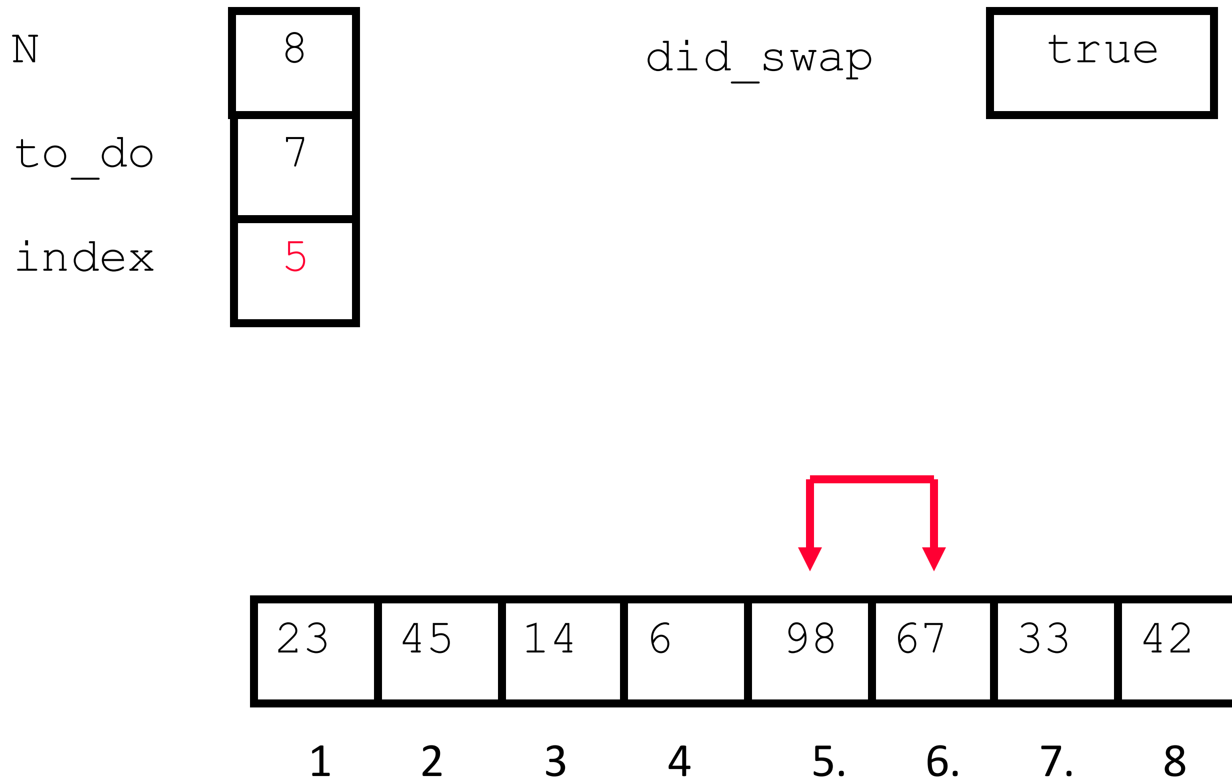
An Animated Example



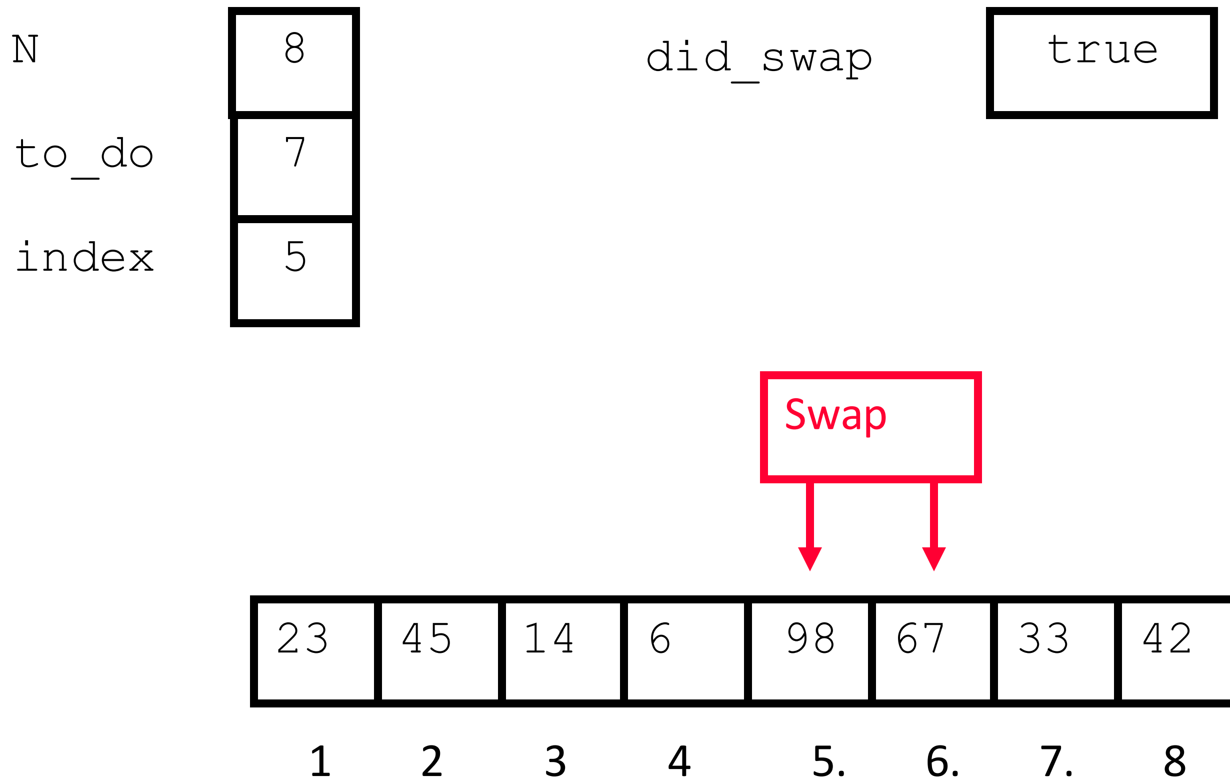
An Animated Example



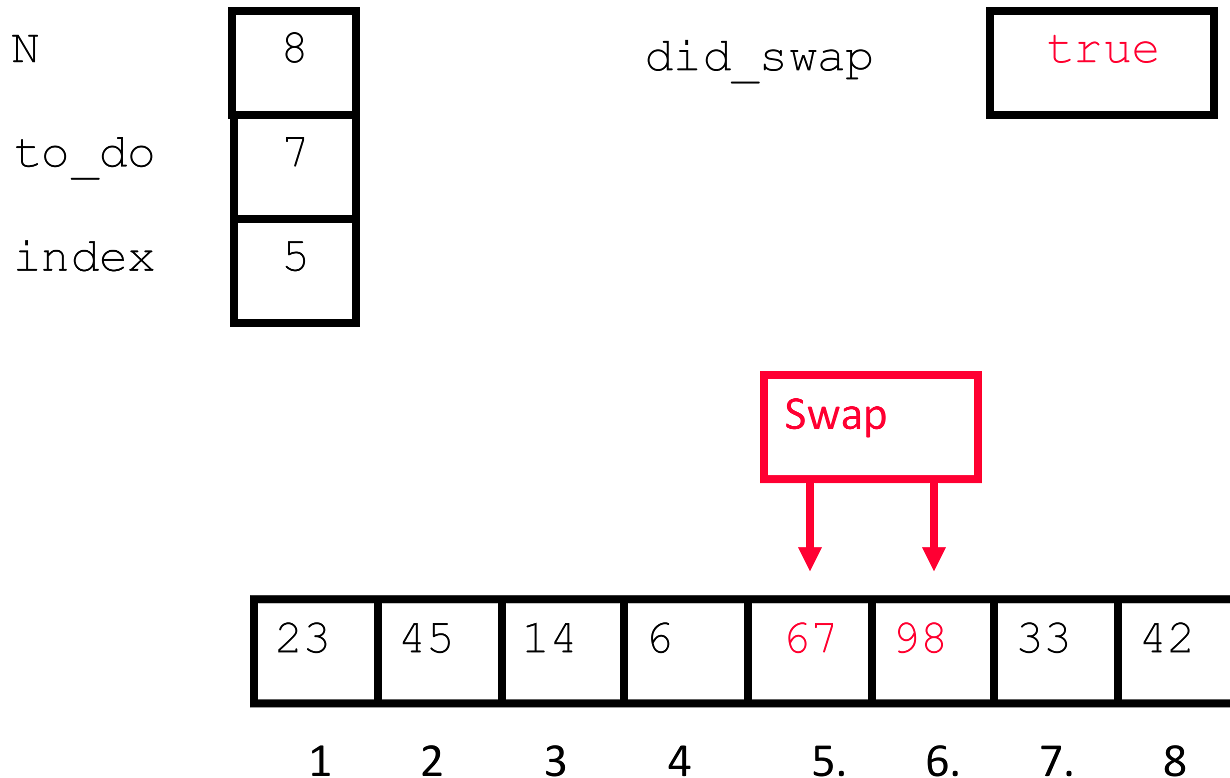
An Animated Example



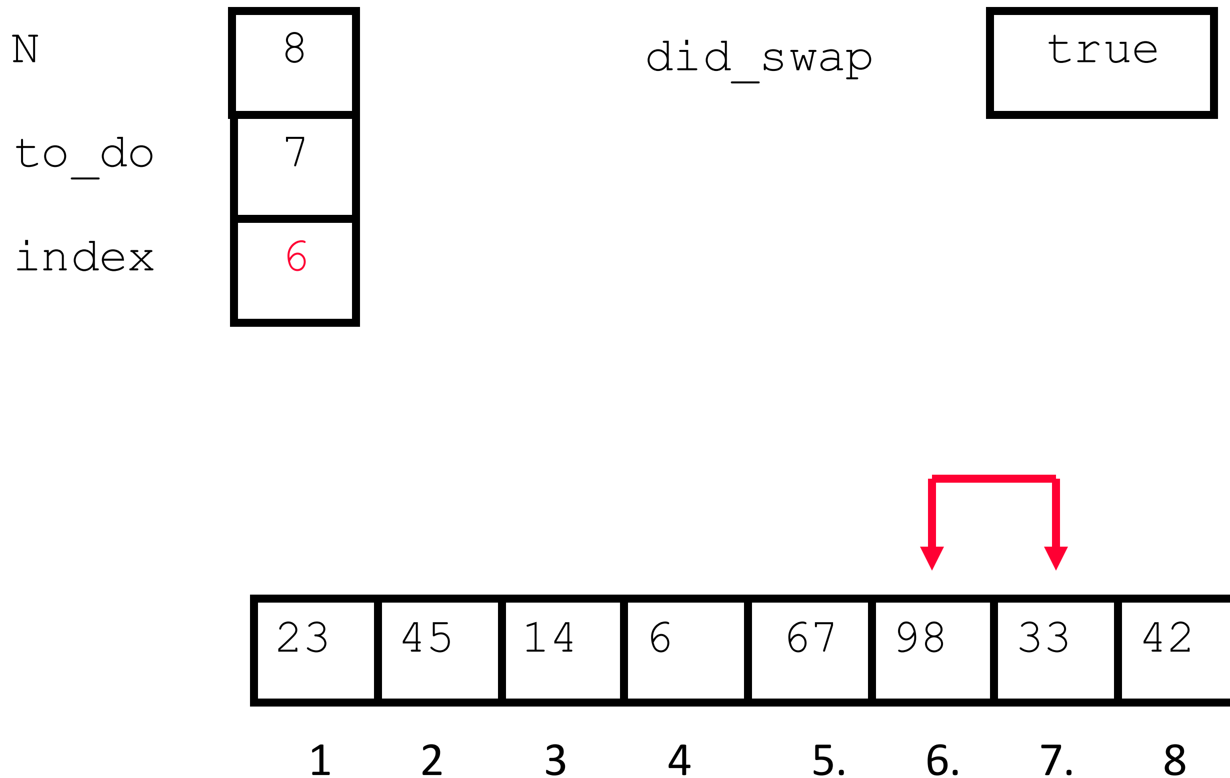
An Animated Example



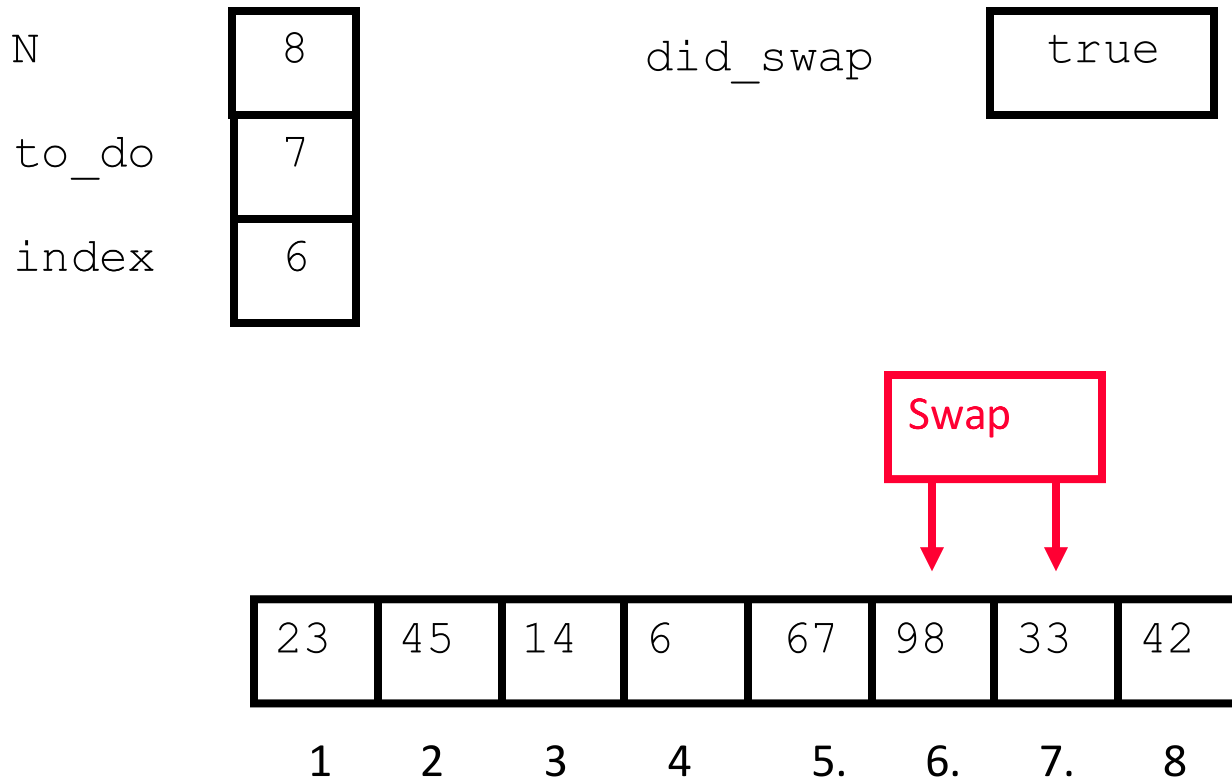
An Animated Example



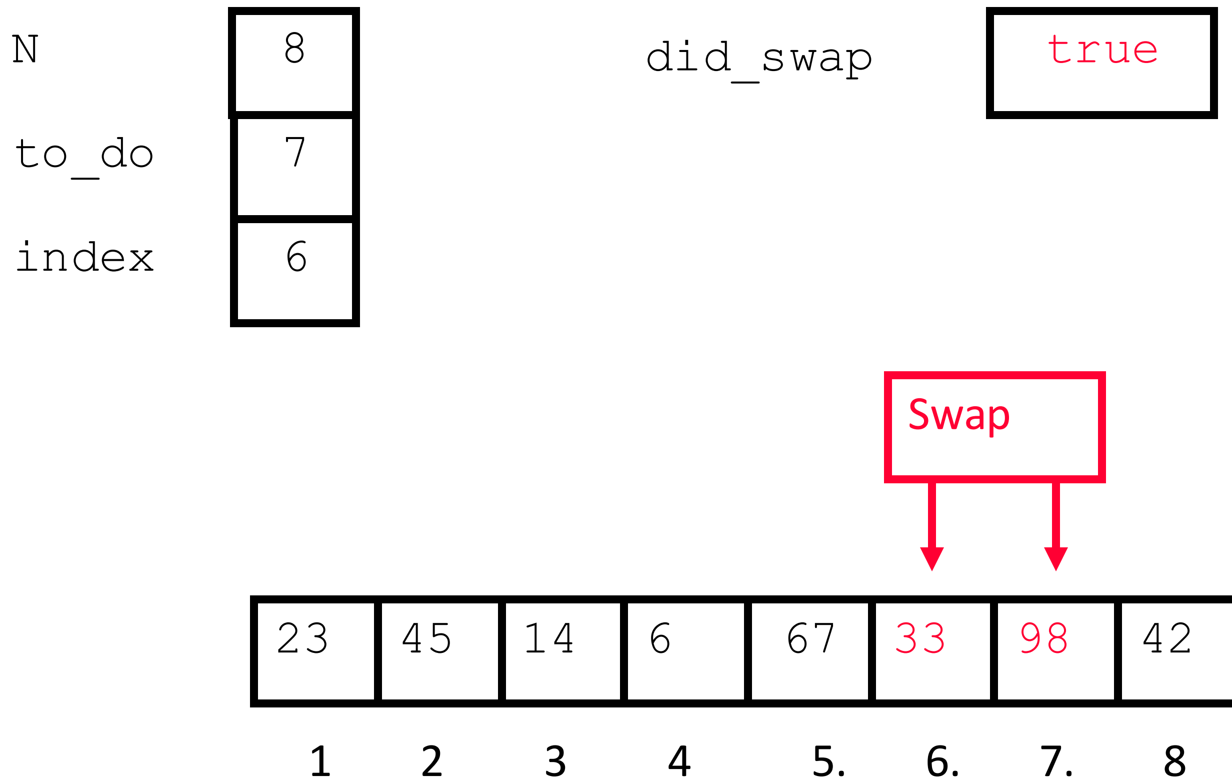
An Animated Example



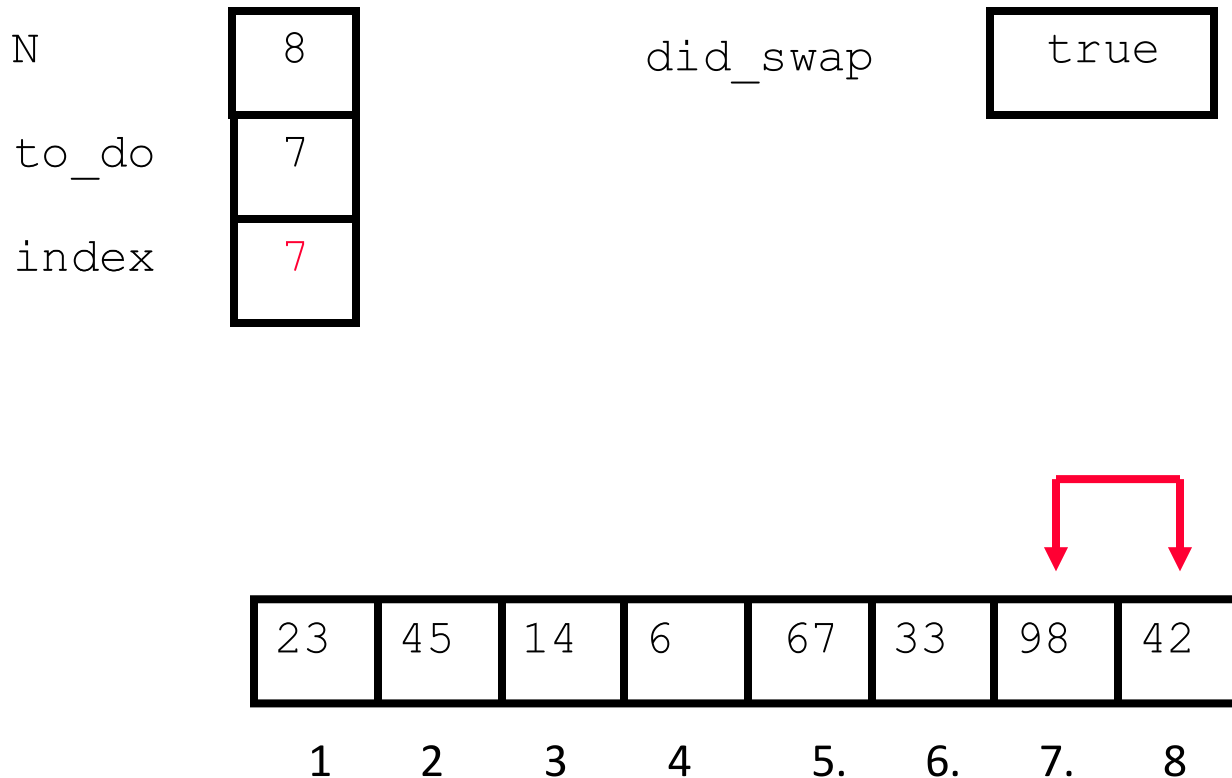
An Animated Example



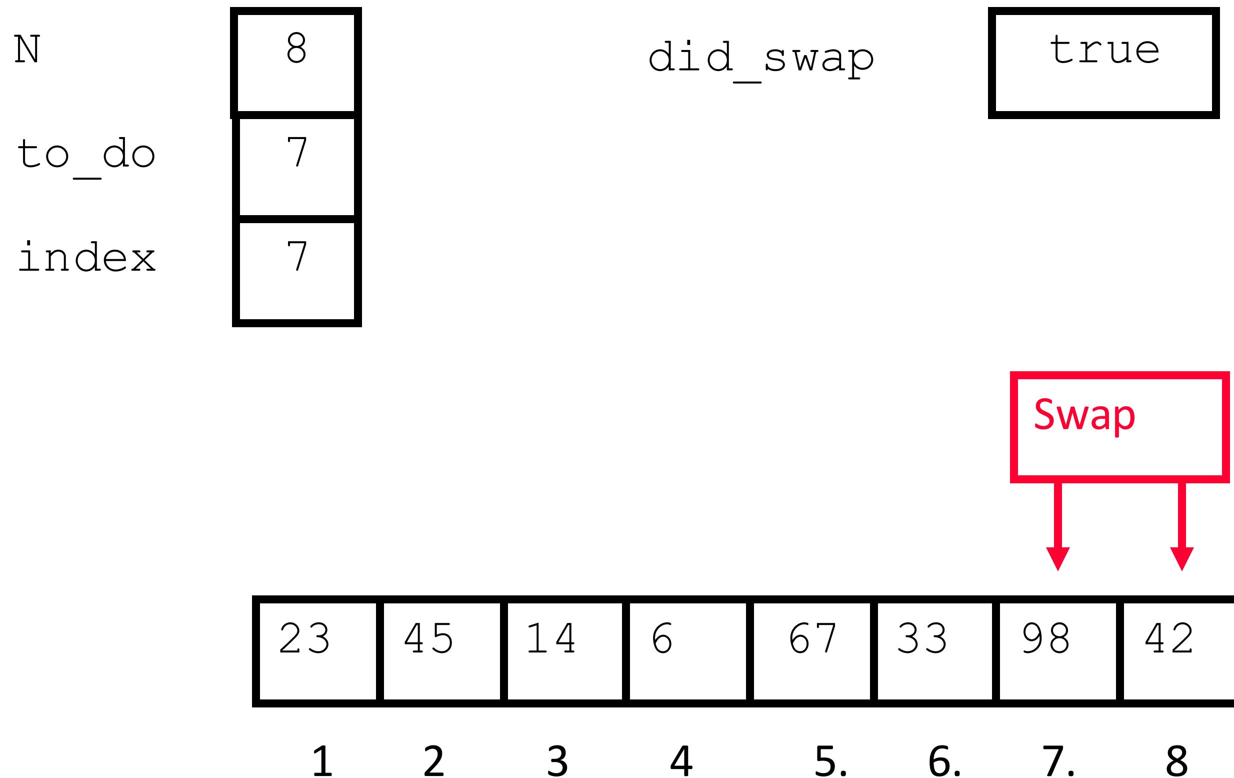
An Animated Example



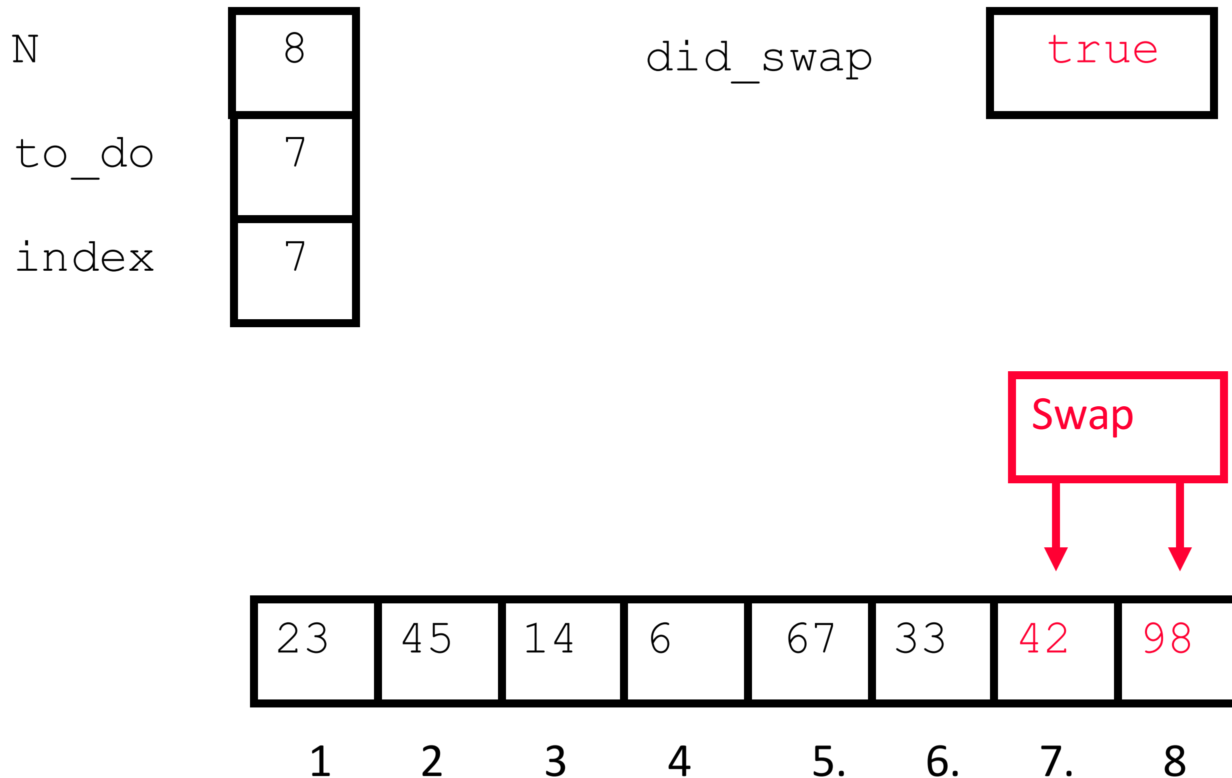
An Animated Example



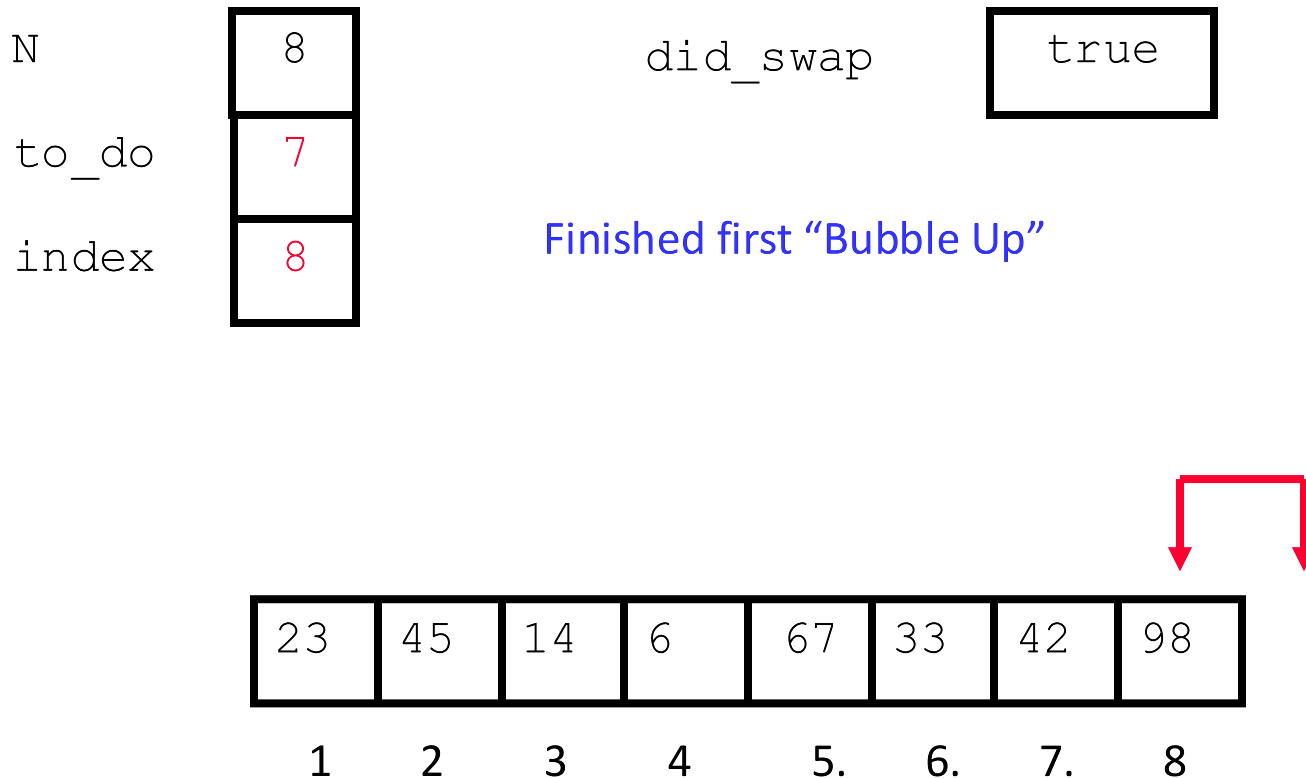
An Animated Example



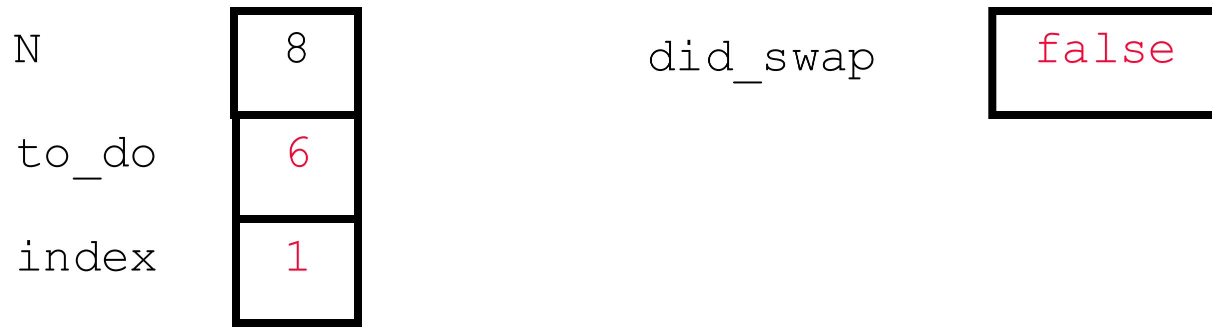
An Animated Example

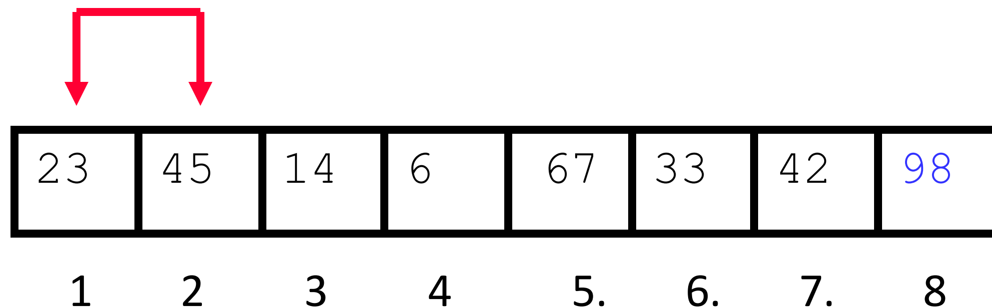


After First Pass of Outer Loop

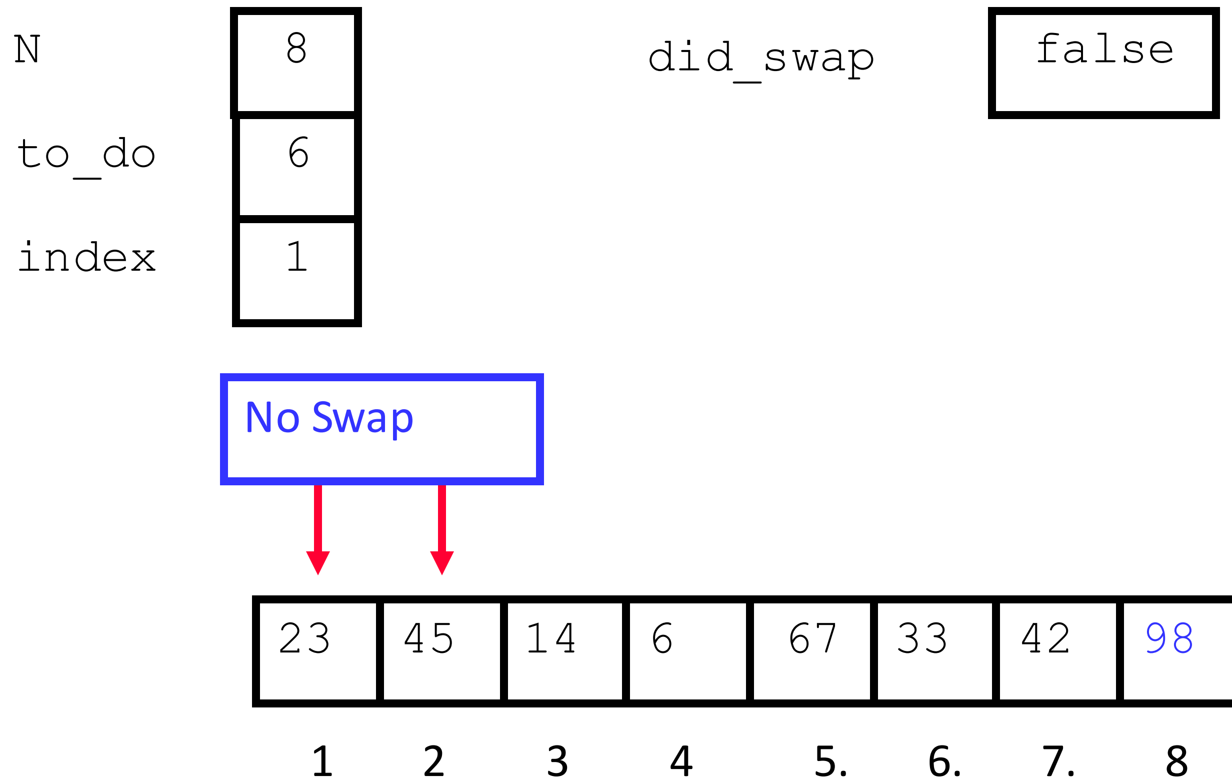


The Second "Bubble Up"

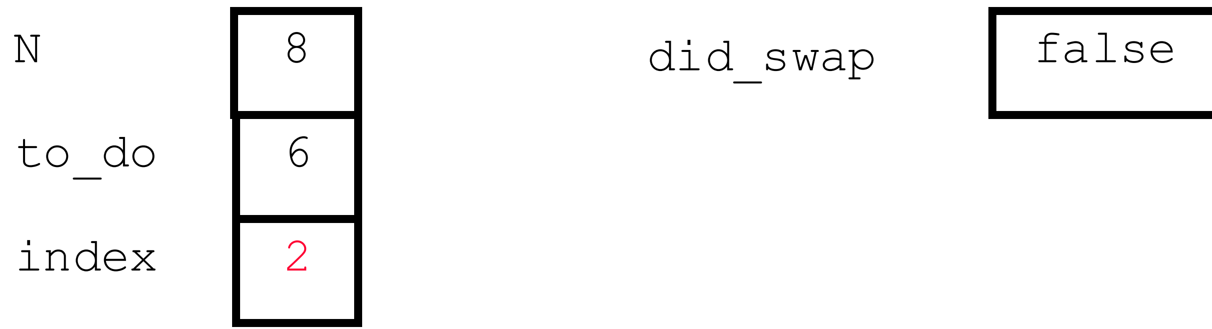


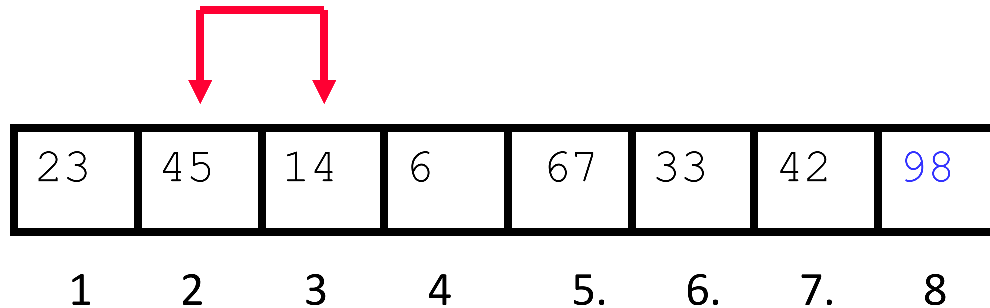


The Second "Bubble Up"

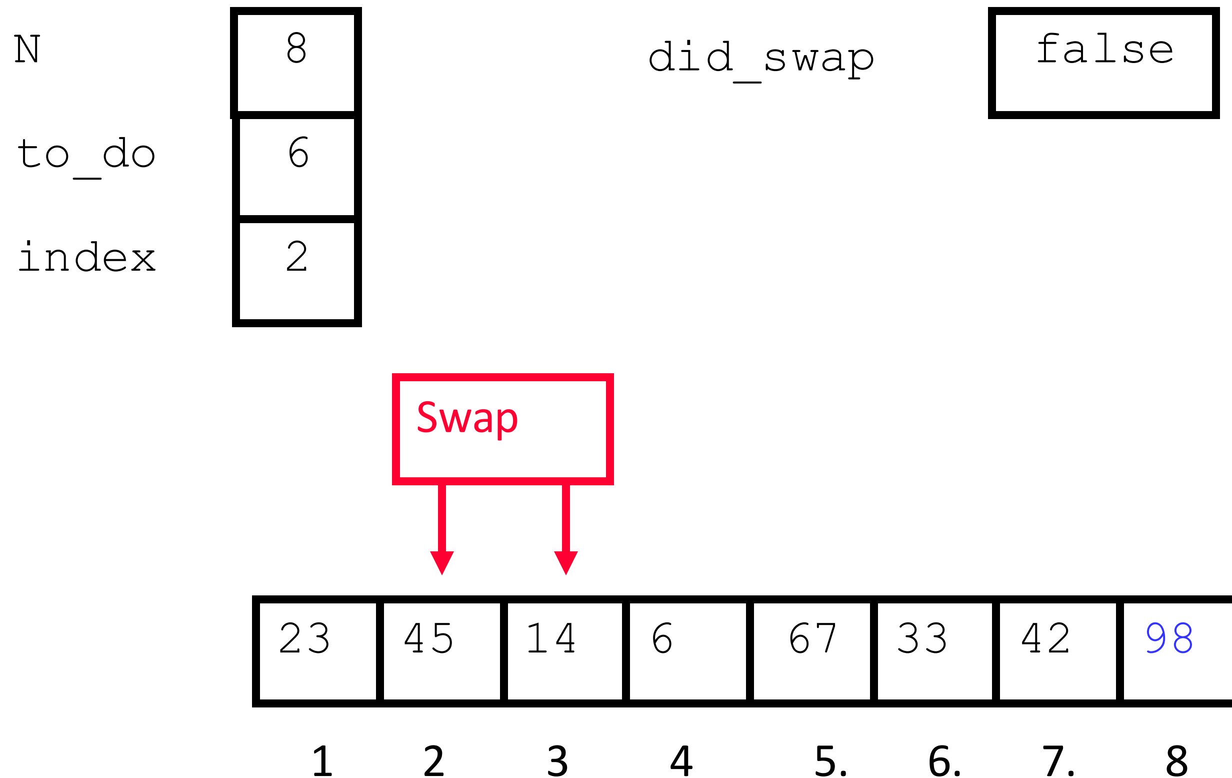


The Second "Bubble Up"

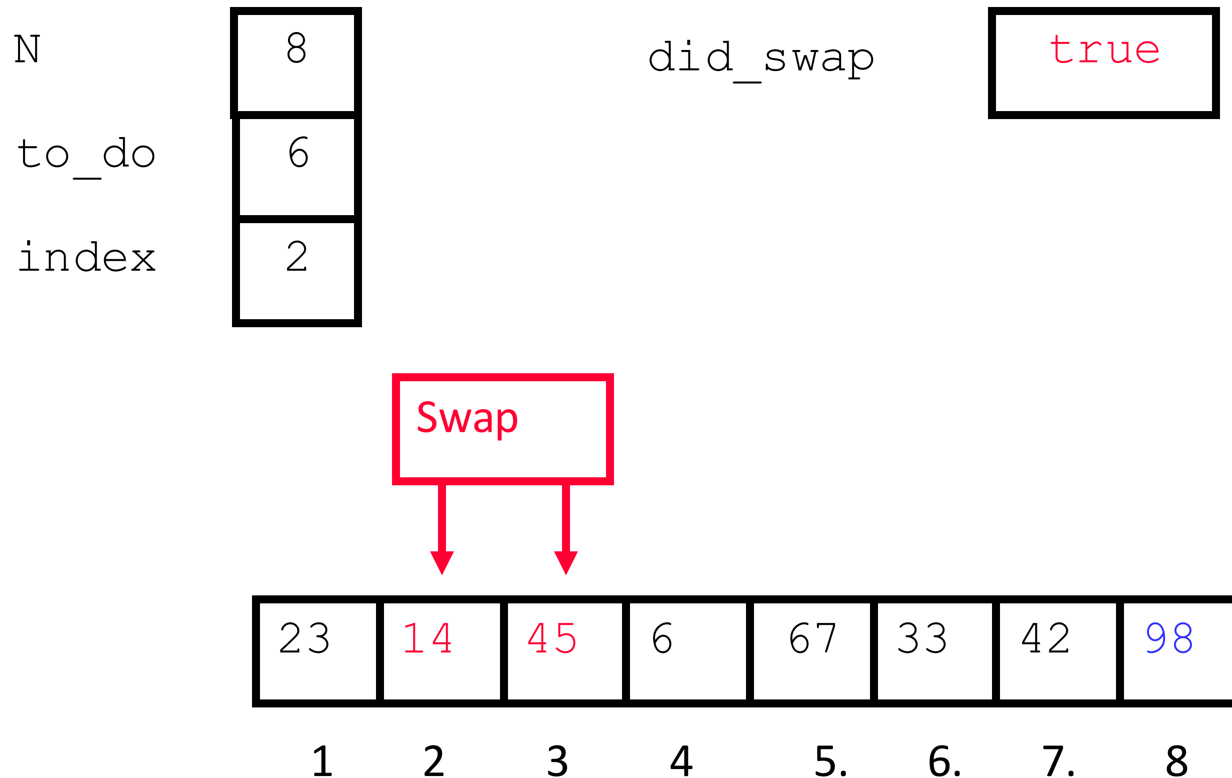




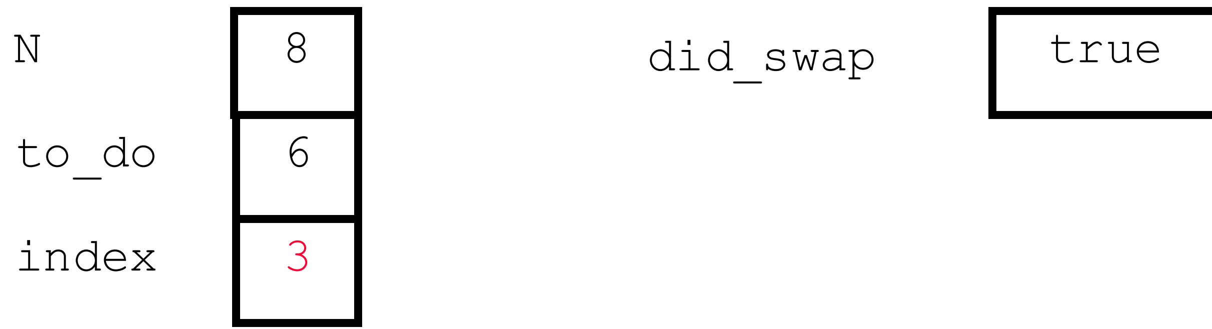
The Second "Bubble Up"

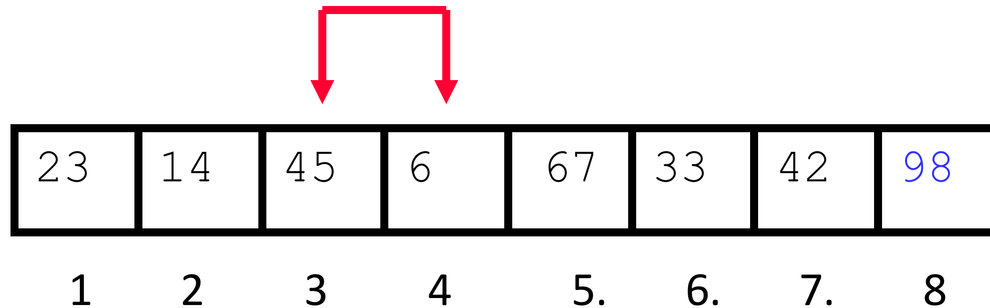


The Second "Bubble Up"

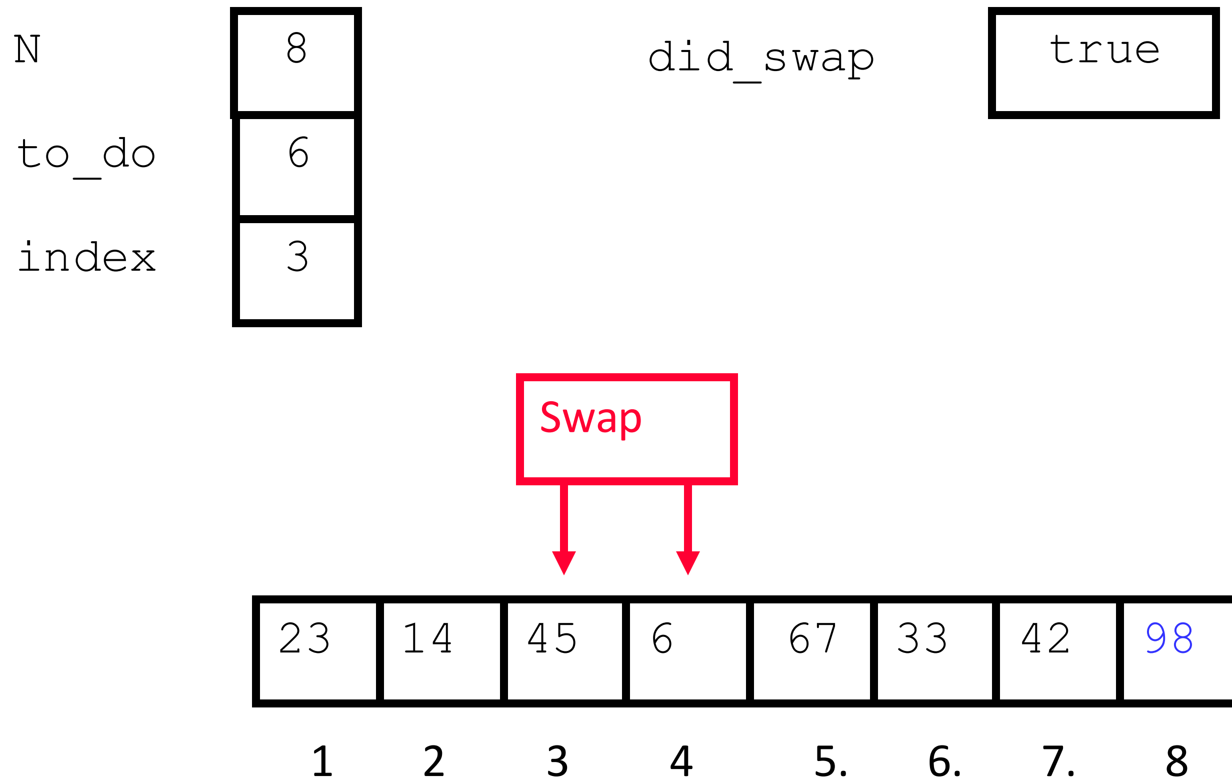


The Second "Bubble Up"

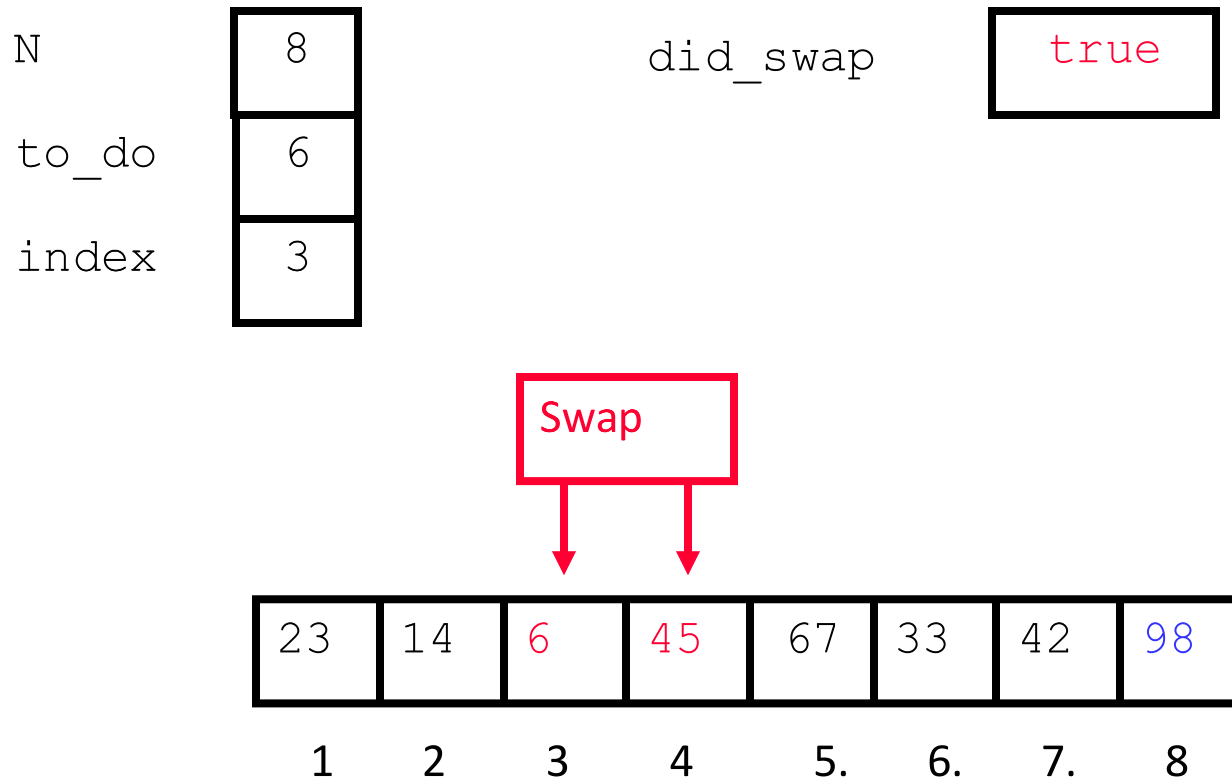




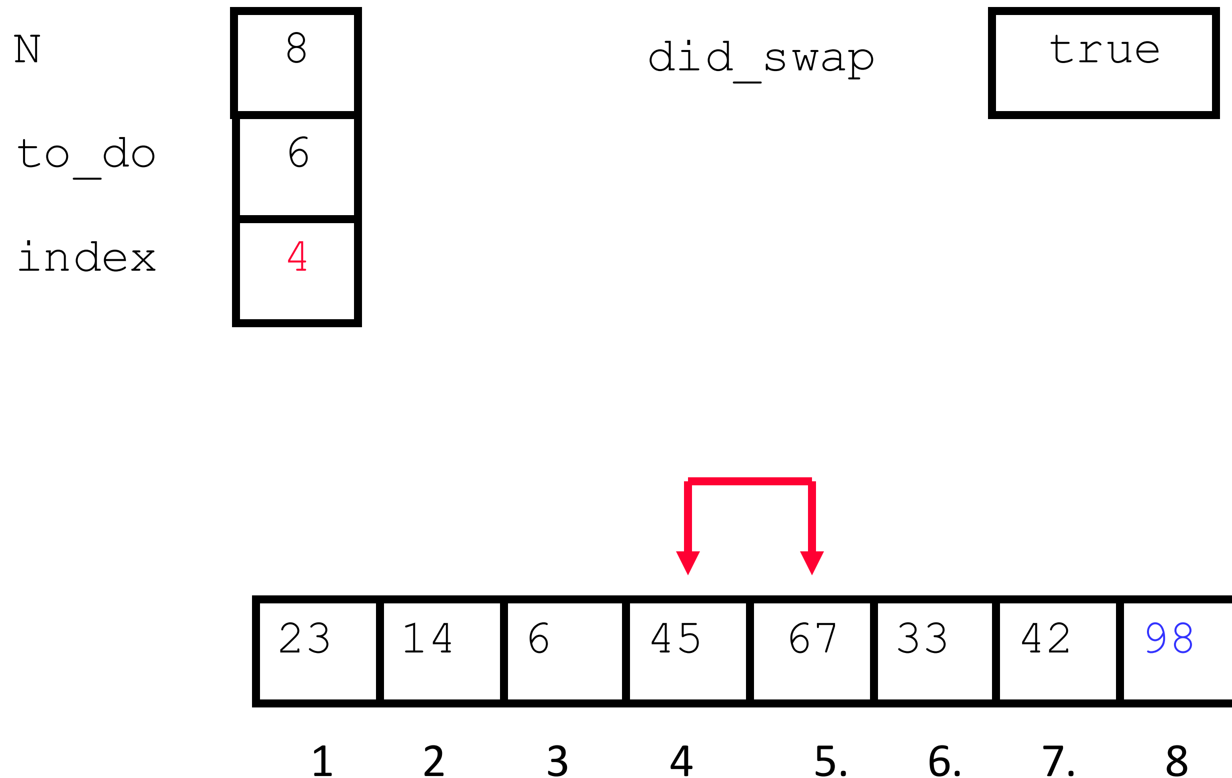
The Second "Bubble Up"



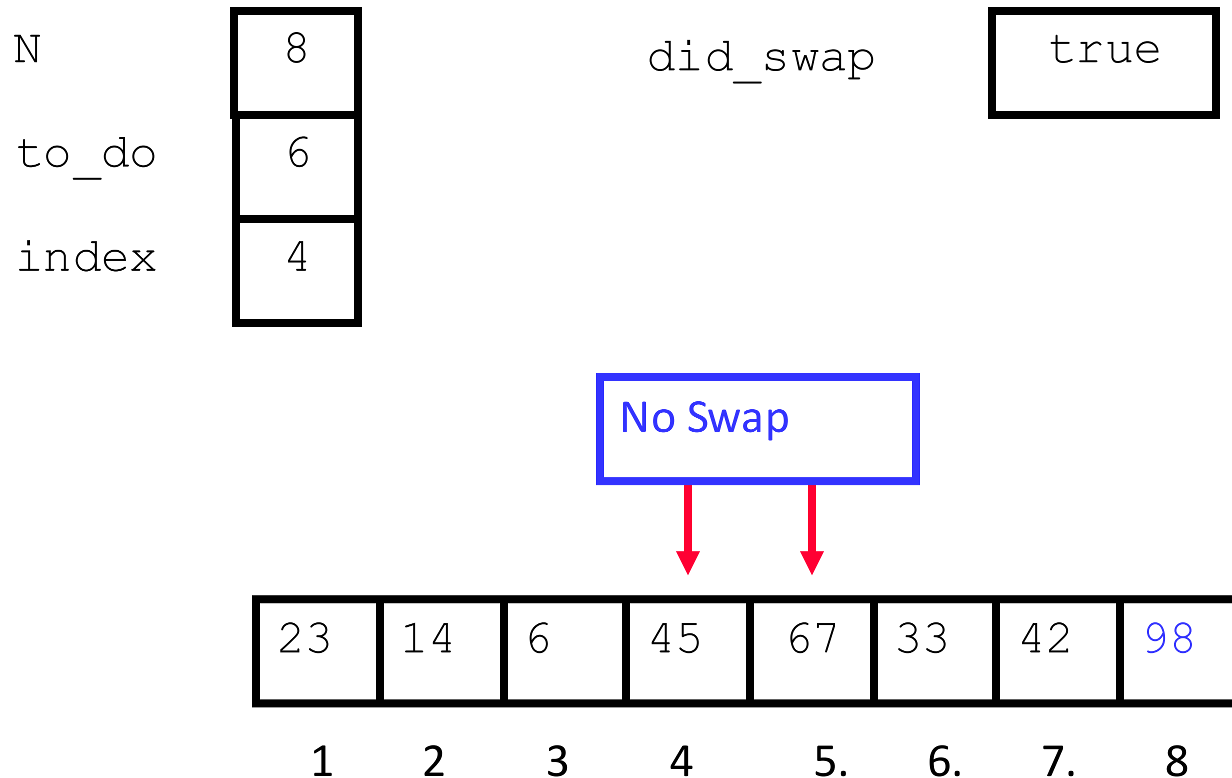
The Second "Bubble Up"



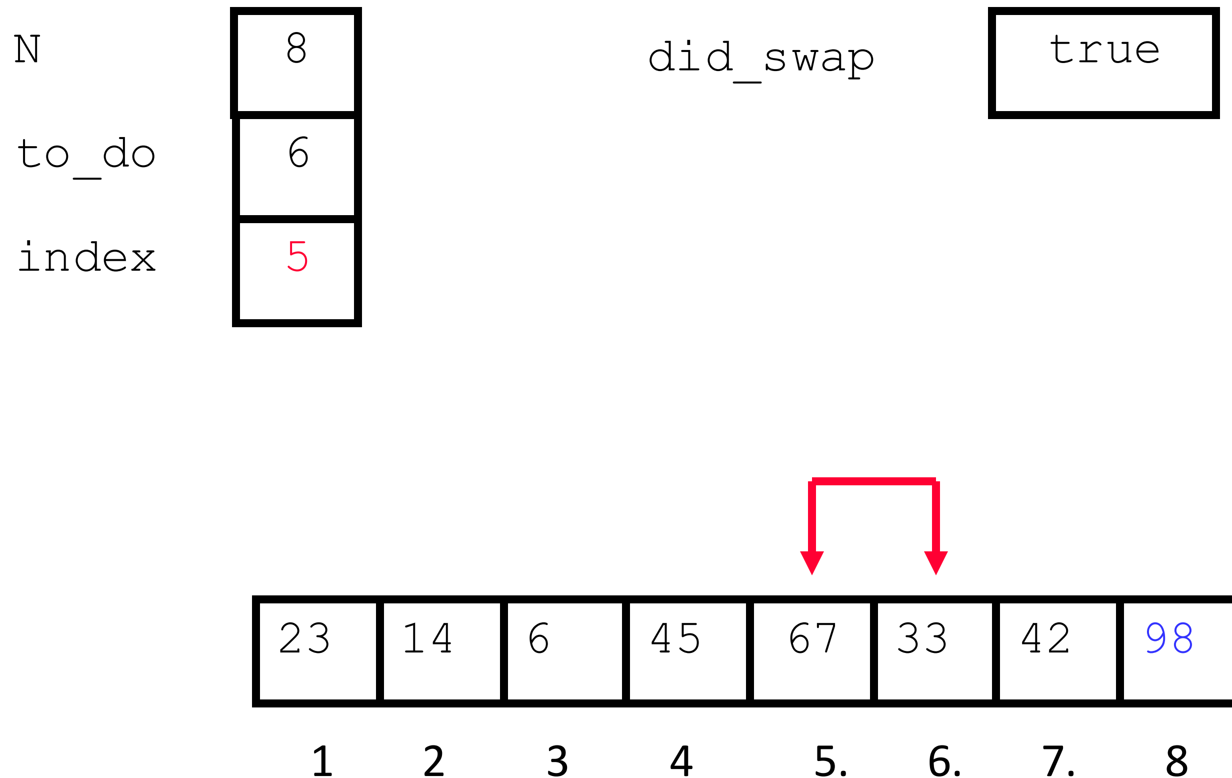
The Second "Bubble Up"



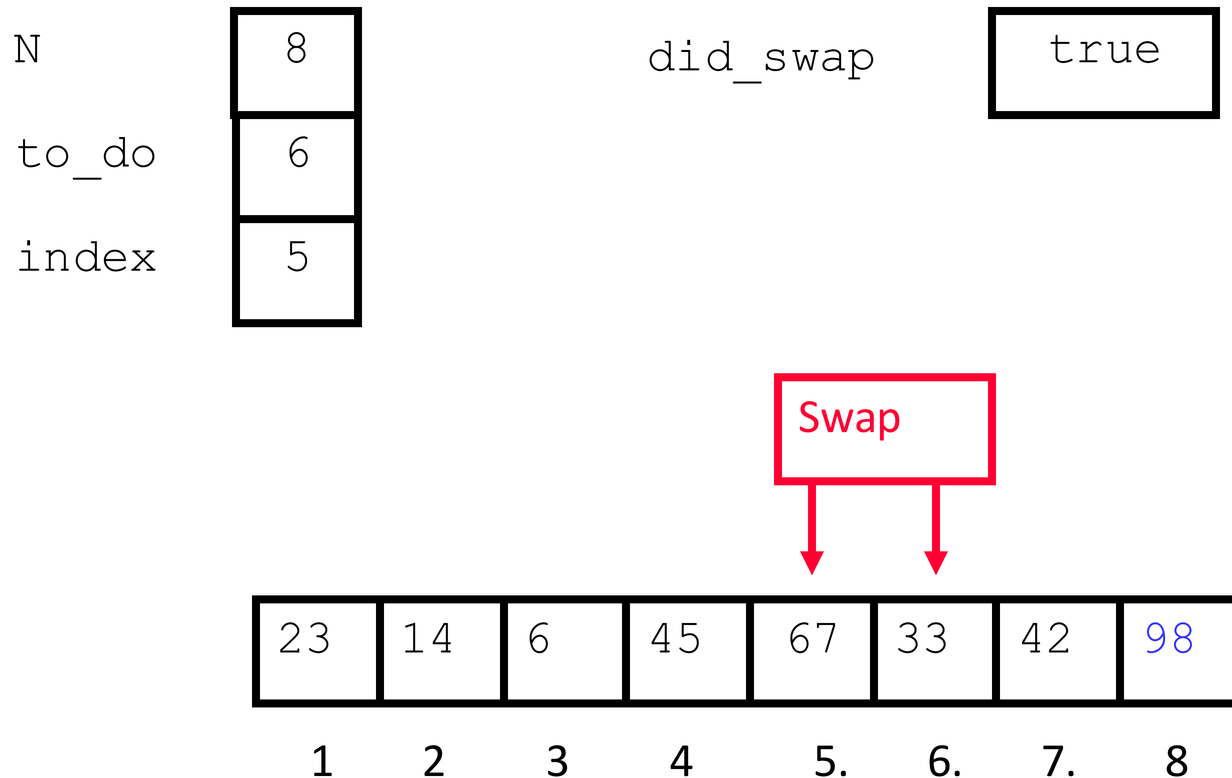
The Second "Bubble Up"



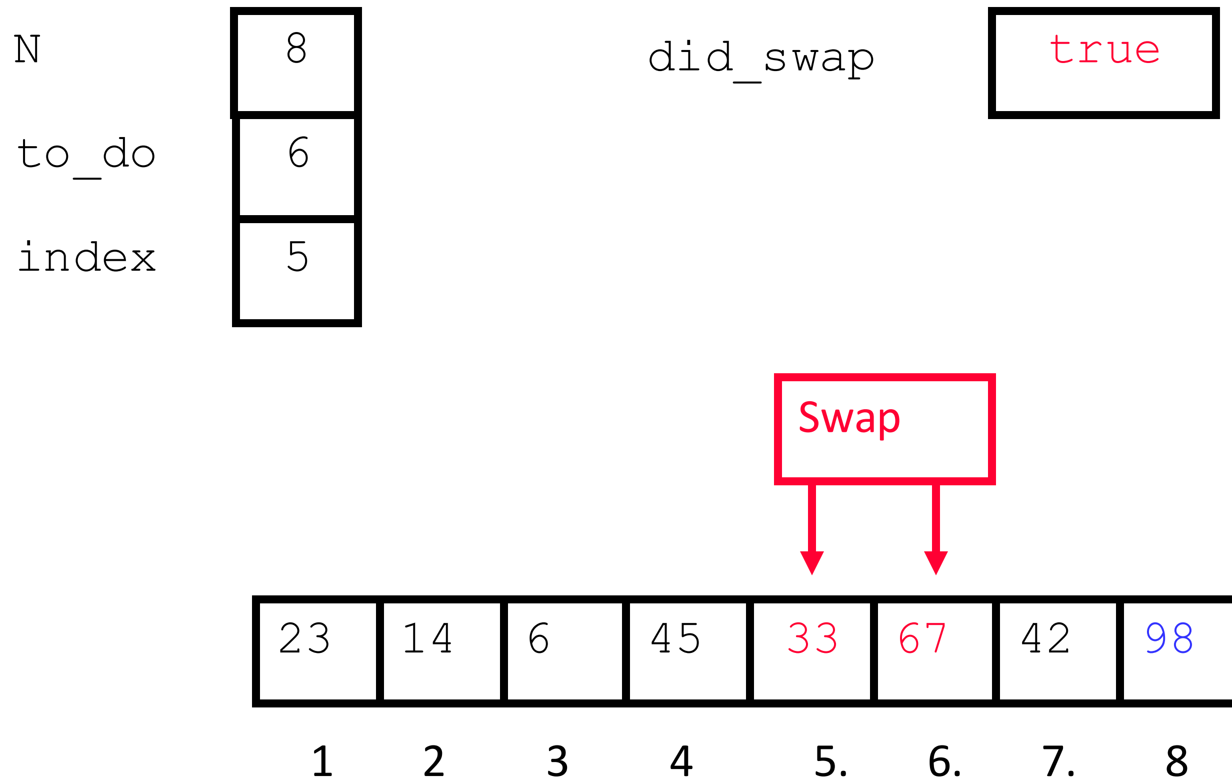
The Second "Bubble Up"



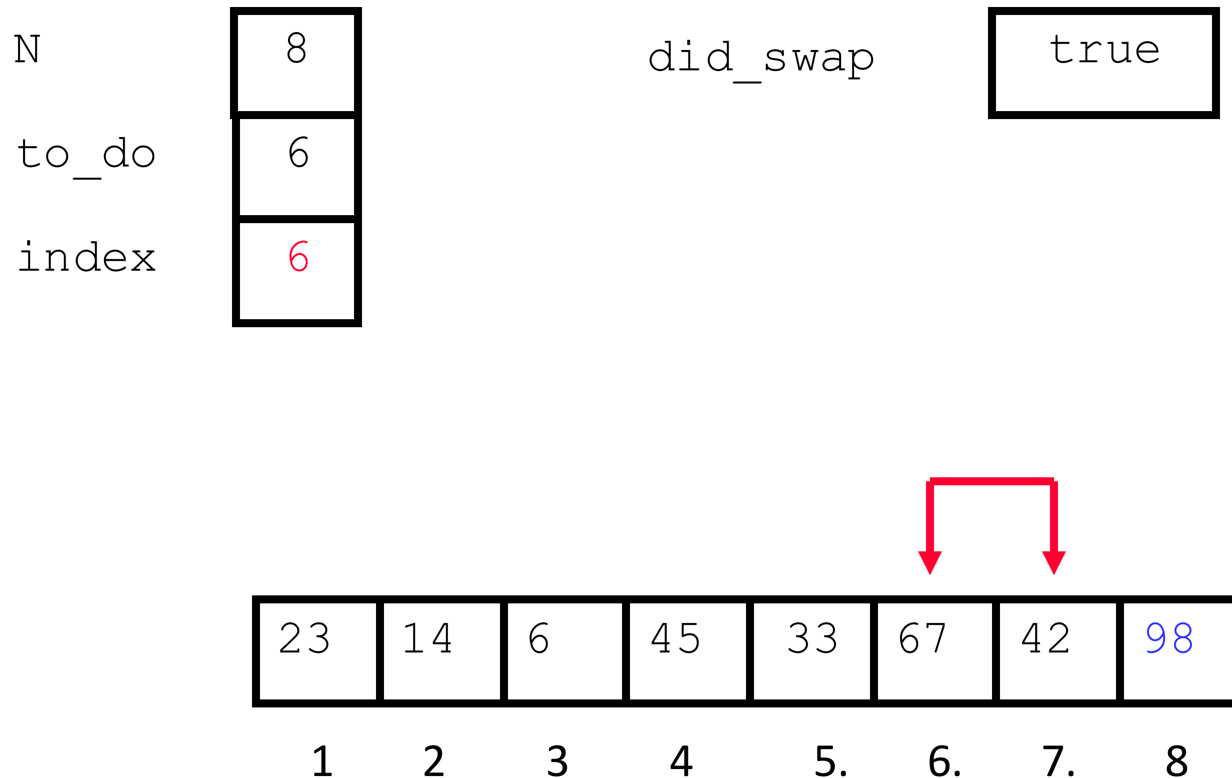
The Second "Bubble Up"



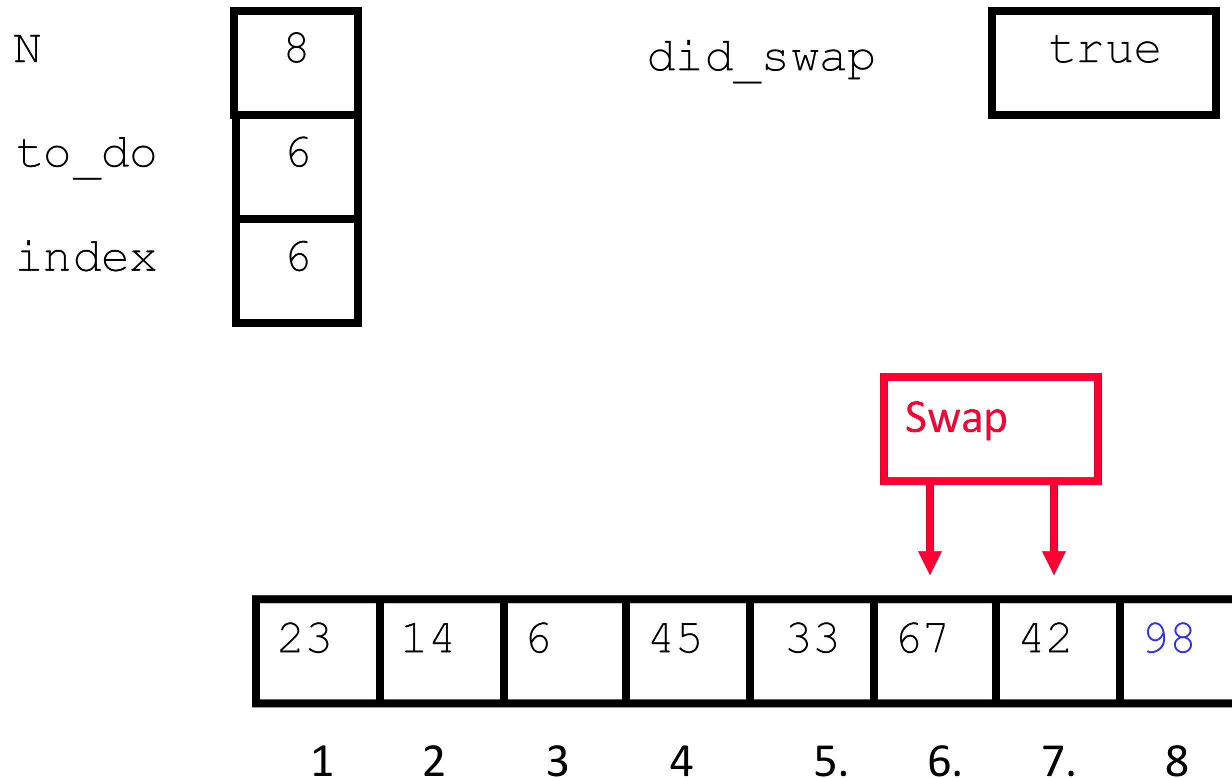
The Second "Bubble Up"



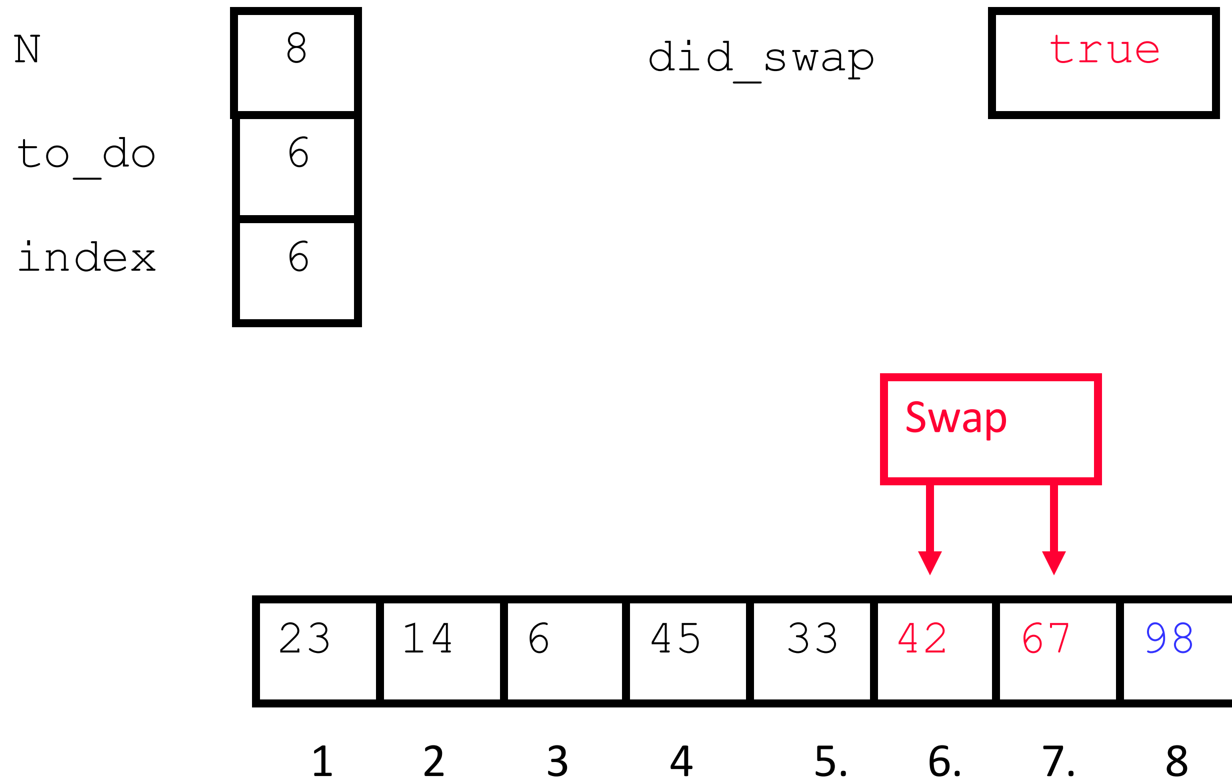
The Second "Bubble Up"



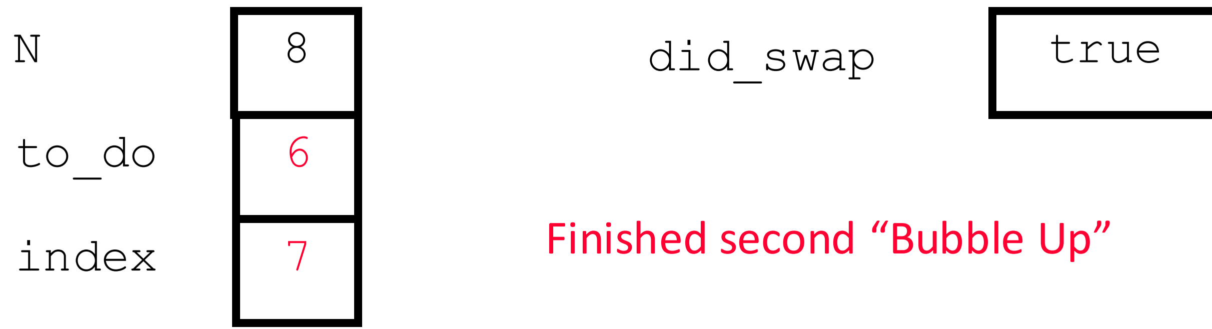
The Second "Bubble Up"

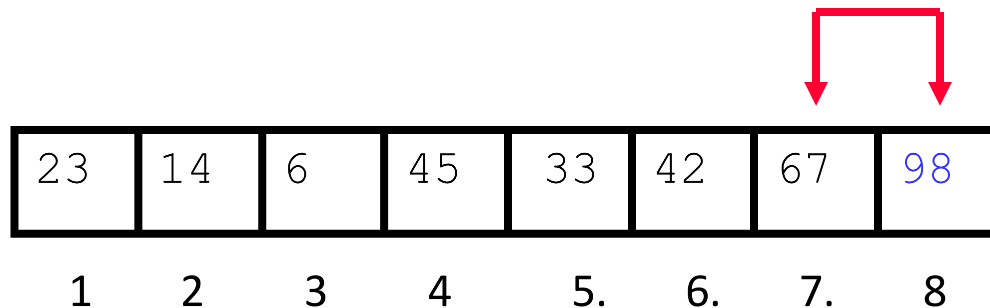


The Second "Bubble Up"

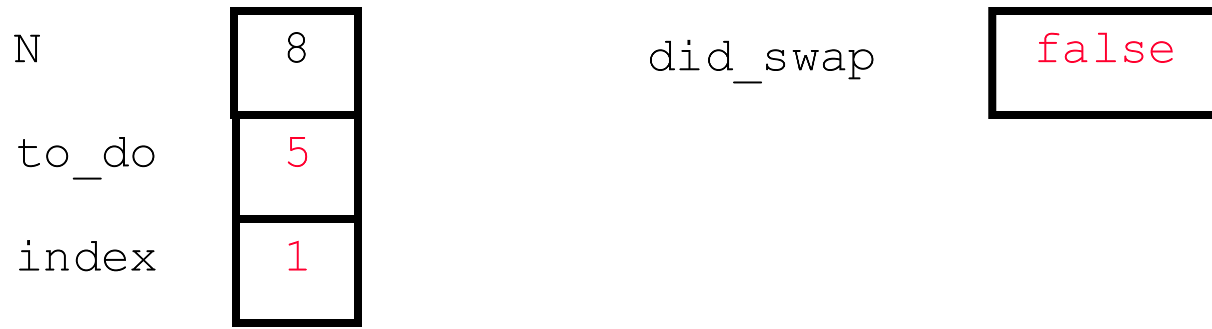


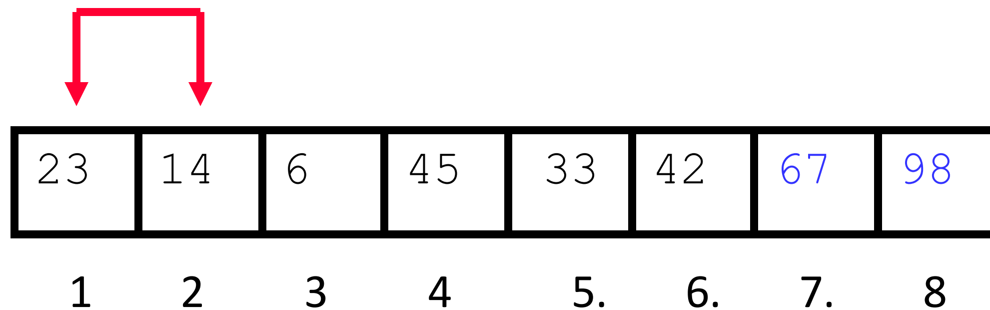
After Second Pass of Outer Loop



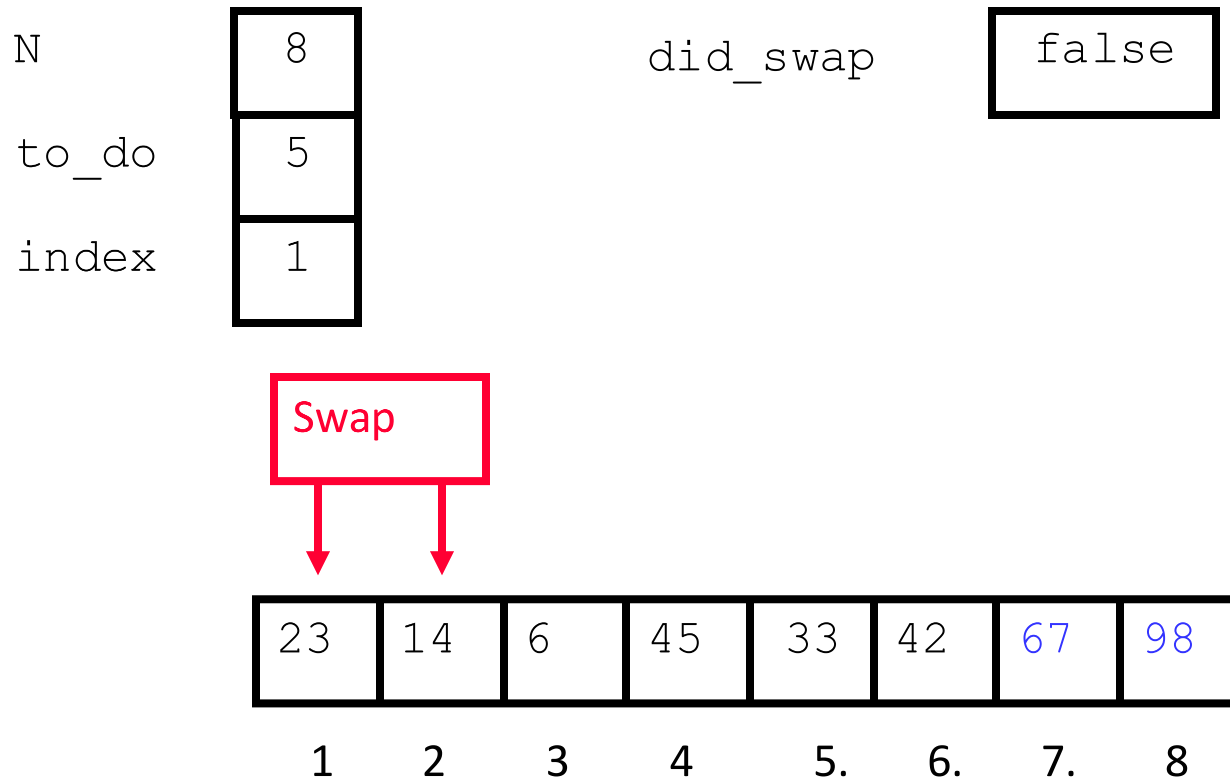


The Third "Bubble Up"

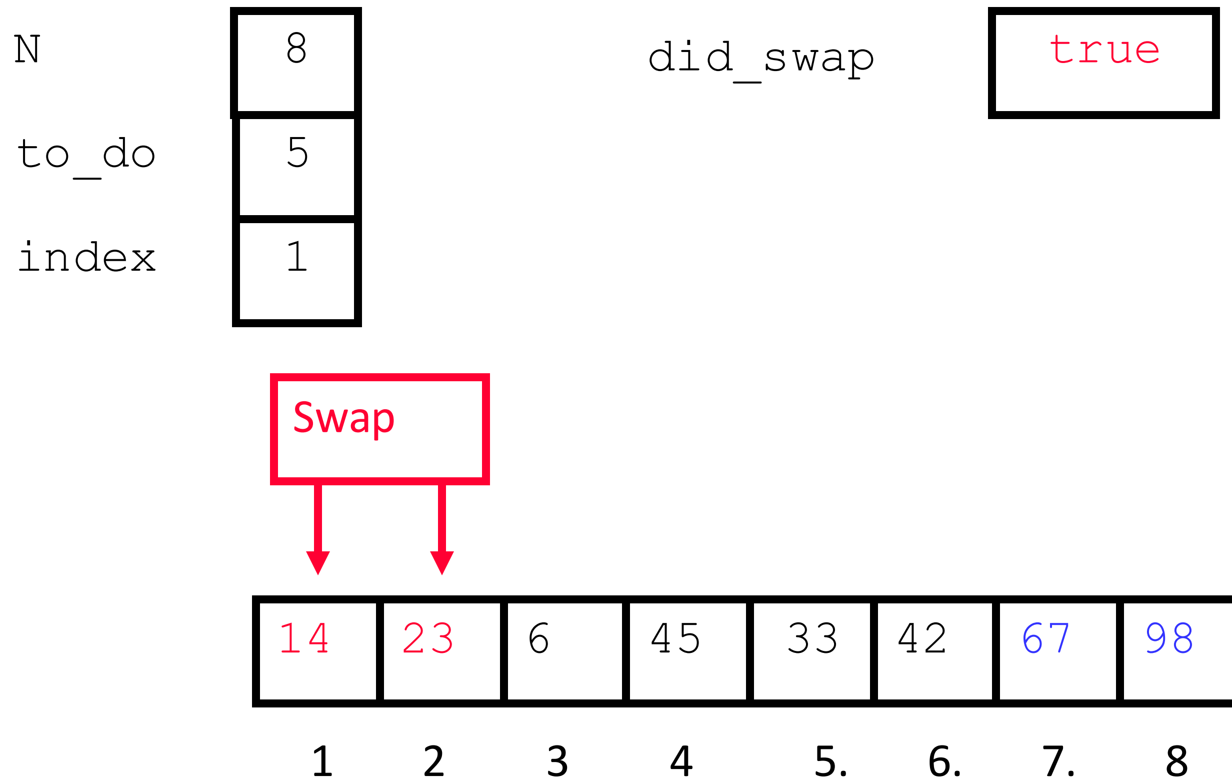




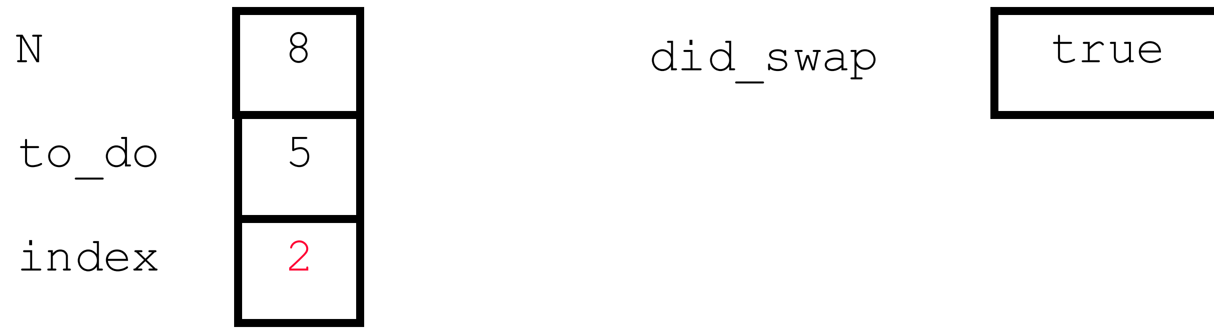
The Third "Bubble Up"

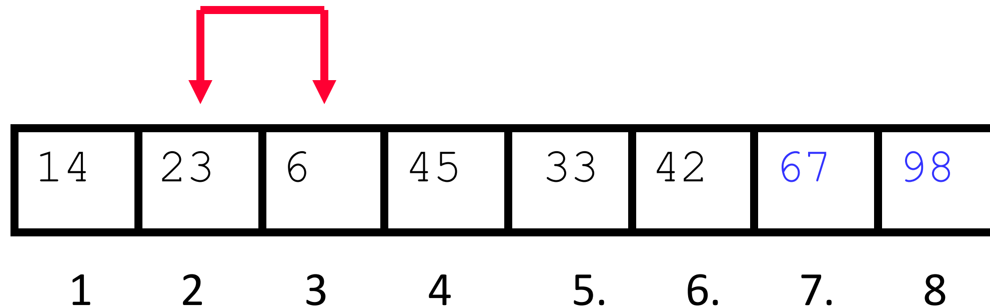


The Third "Bubble Up"

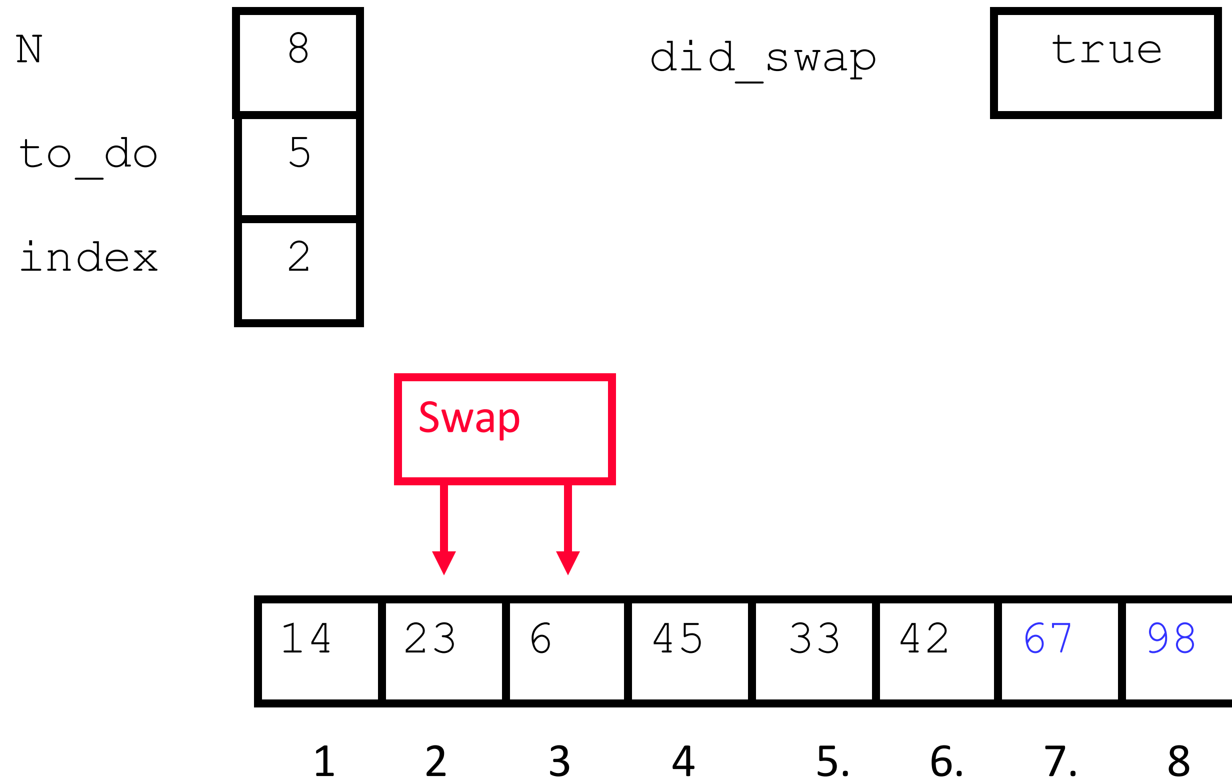


The Third "Bubble Up"

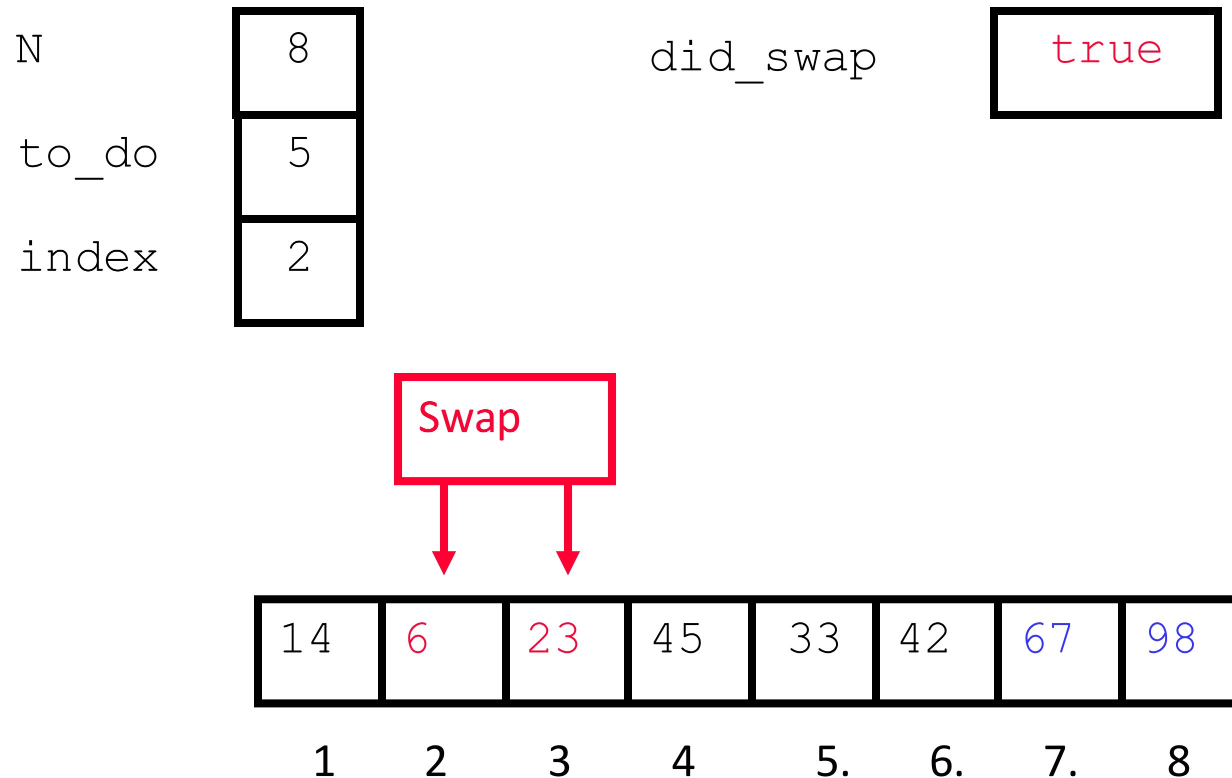




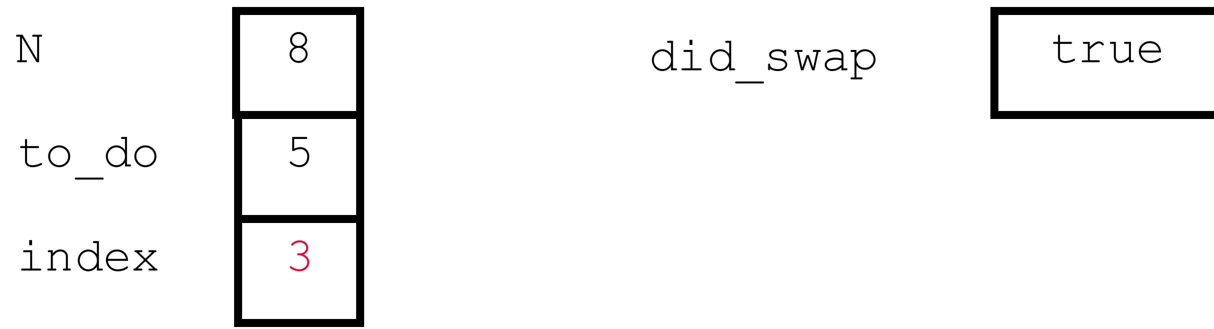
The Third "Bubble Up"

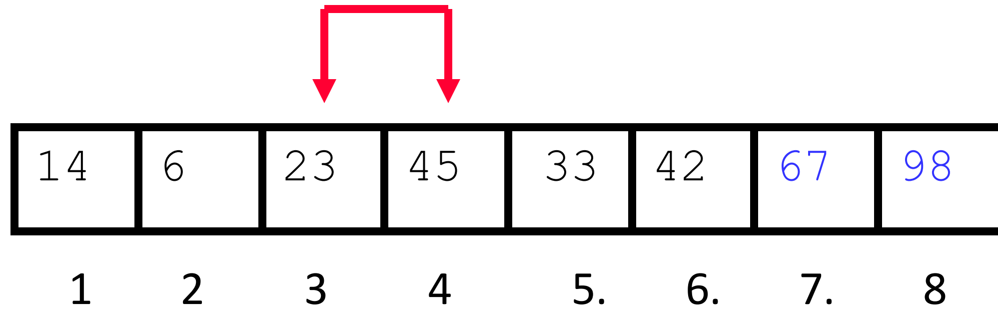


The Third "Bubble Up"

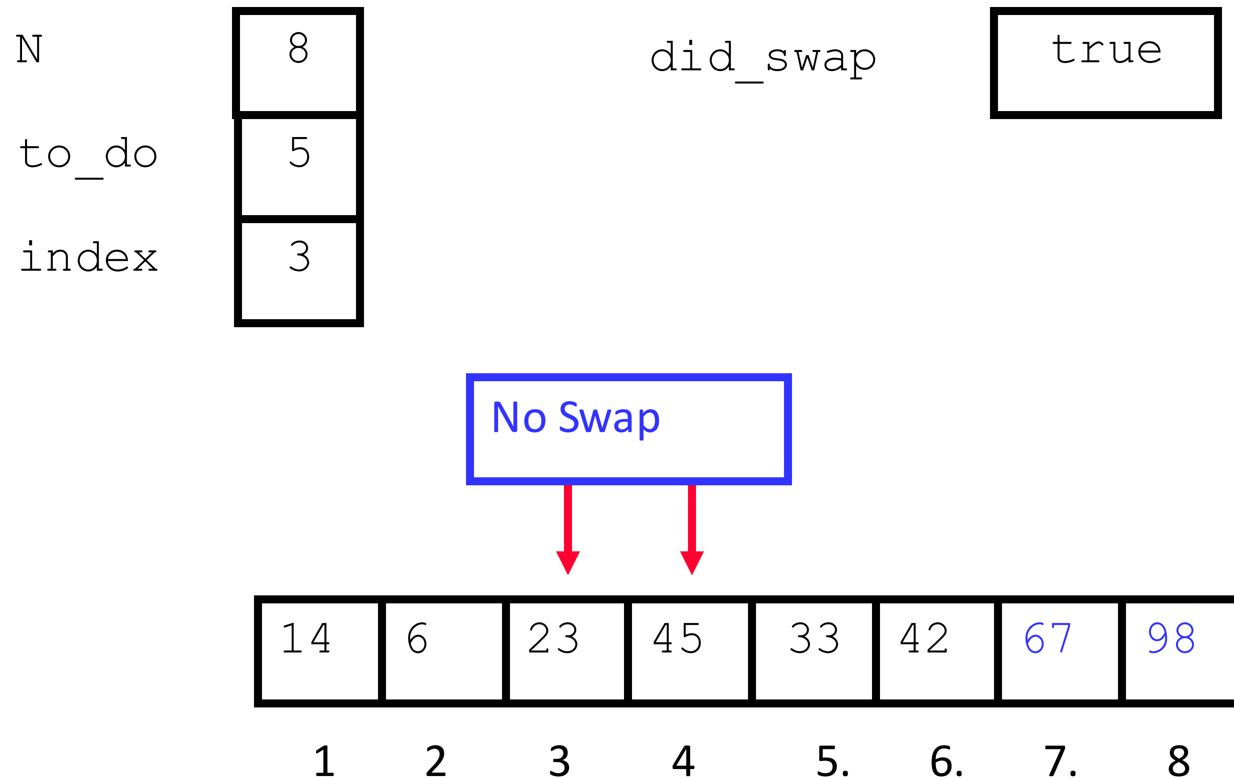


The Third “Bubble Up”

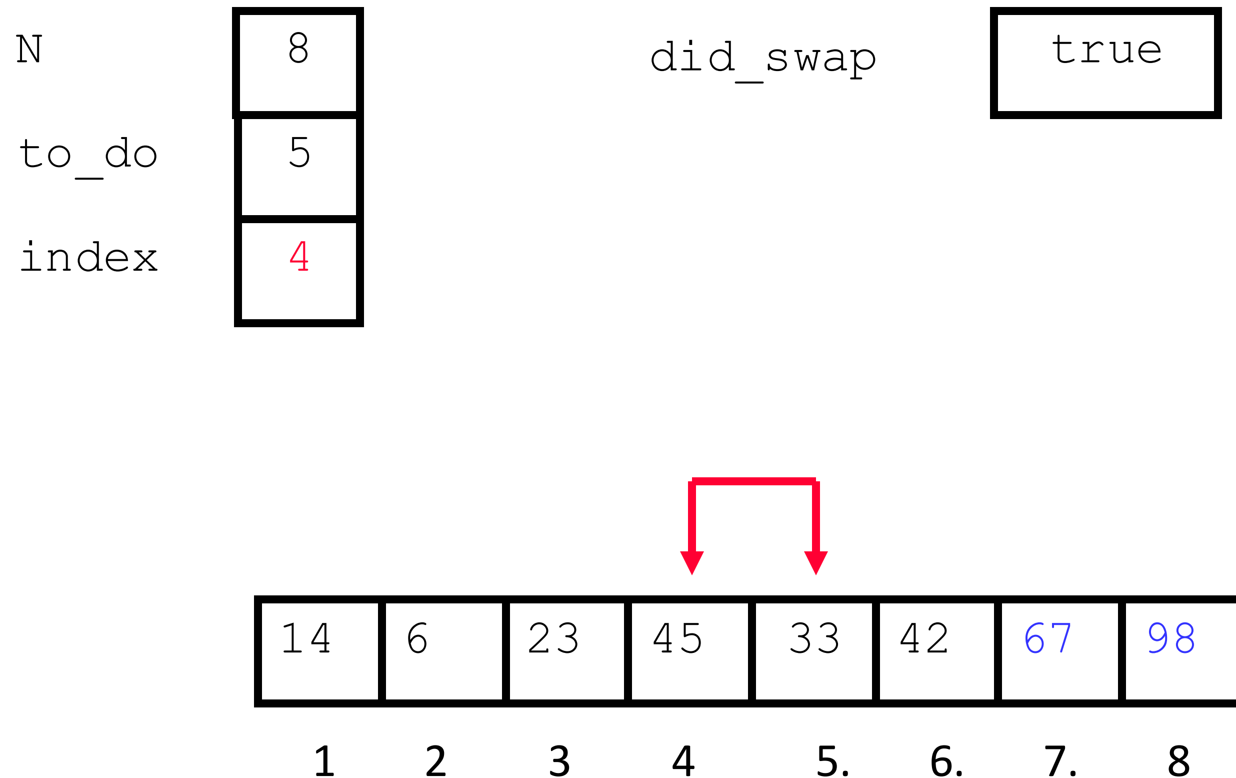




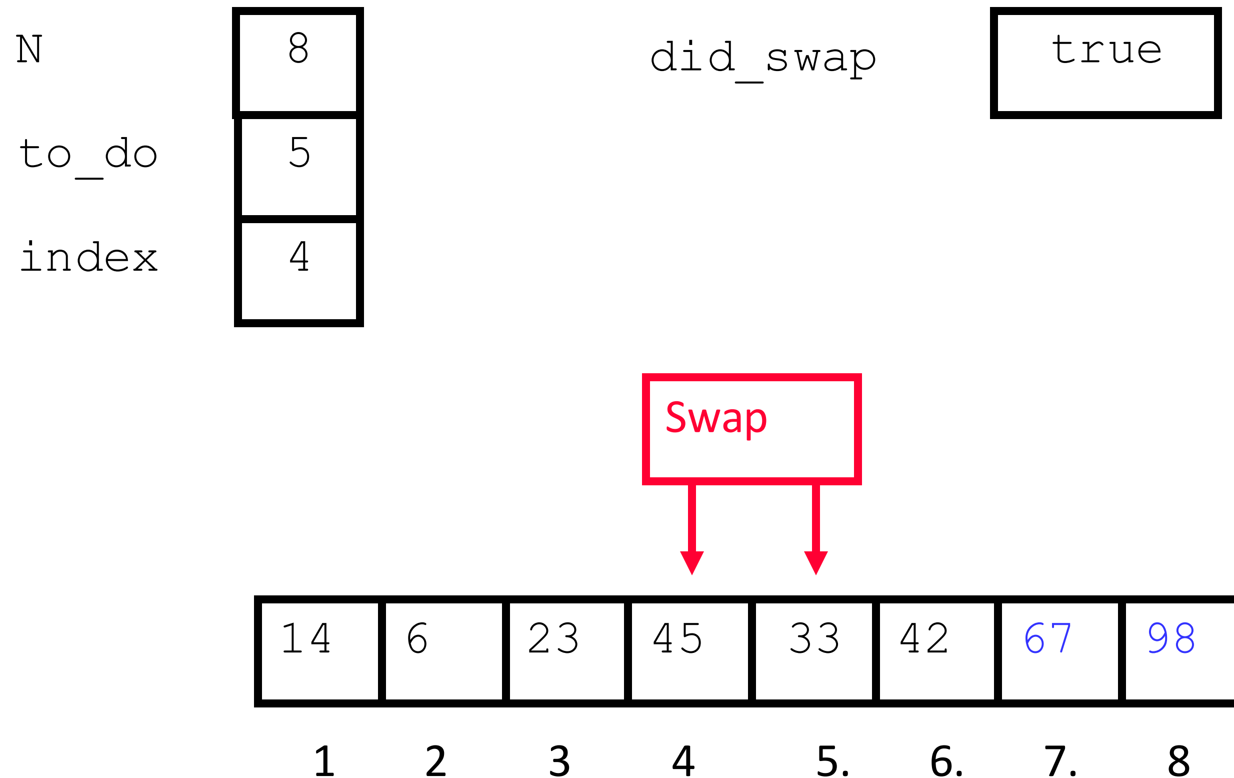
The Third "Bubble Up"



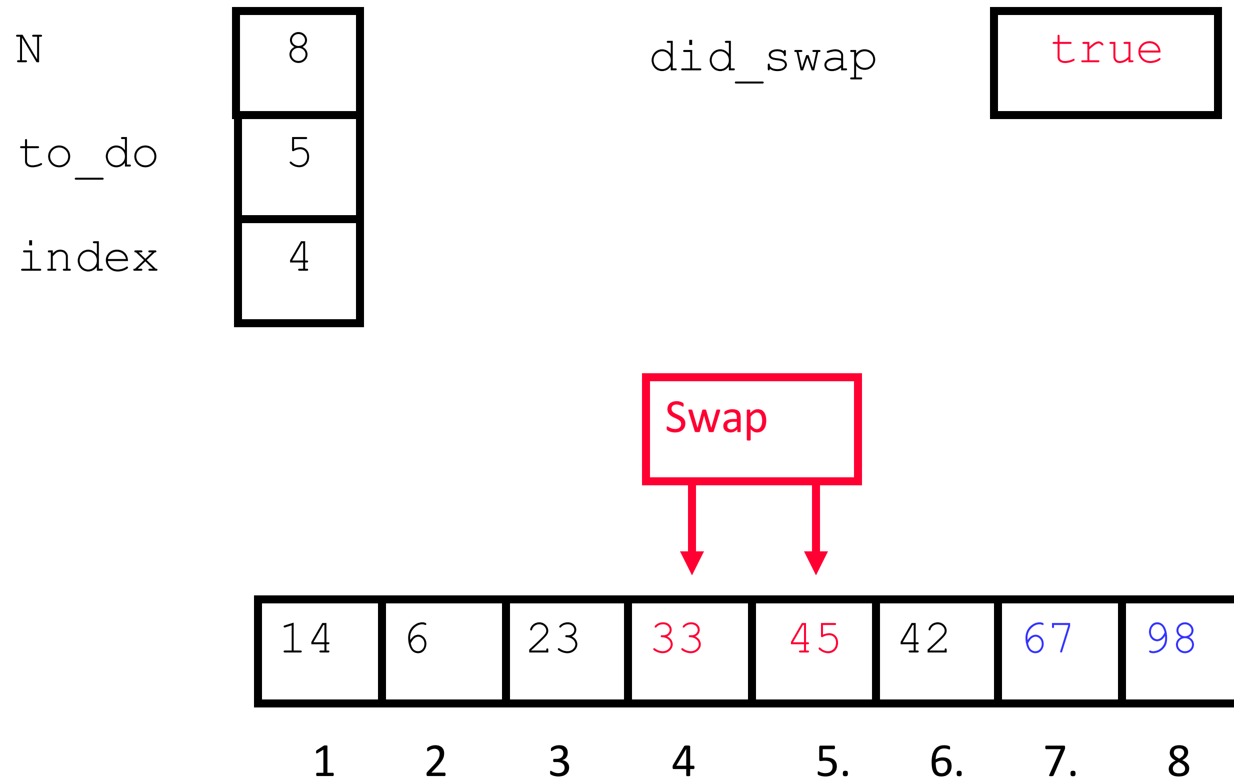
The Third "Bubble Up"



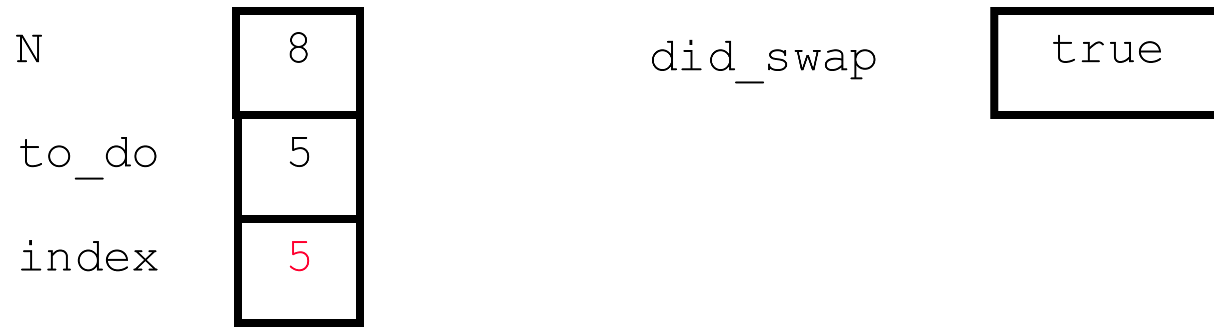
The Third "Bubble Up"

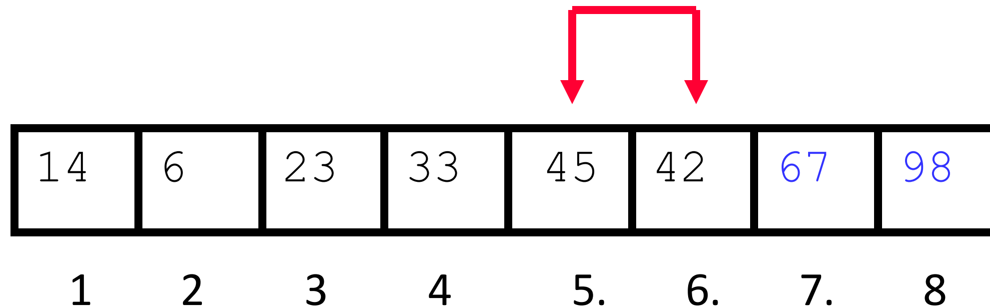


The Third "Bubble Up"

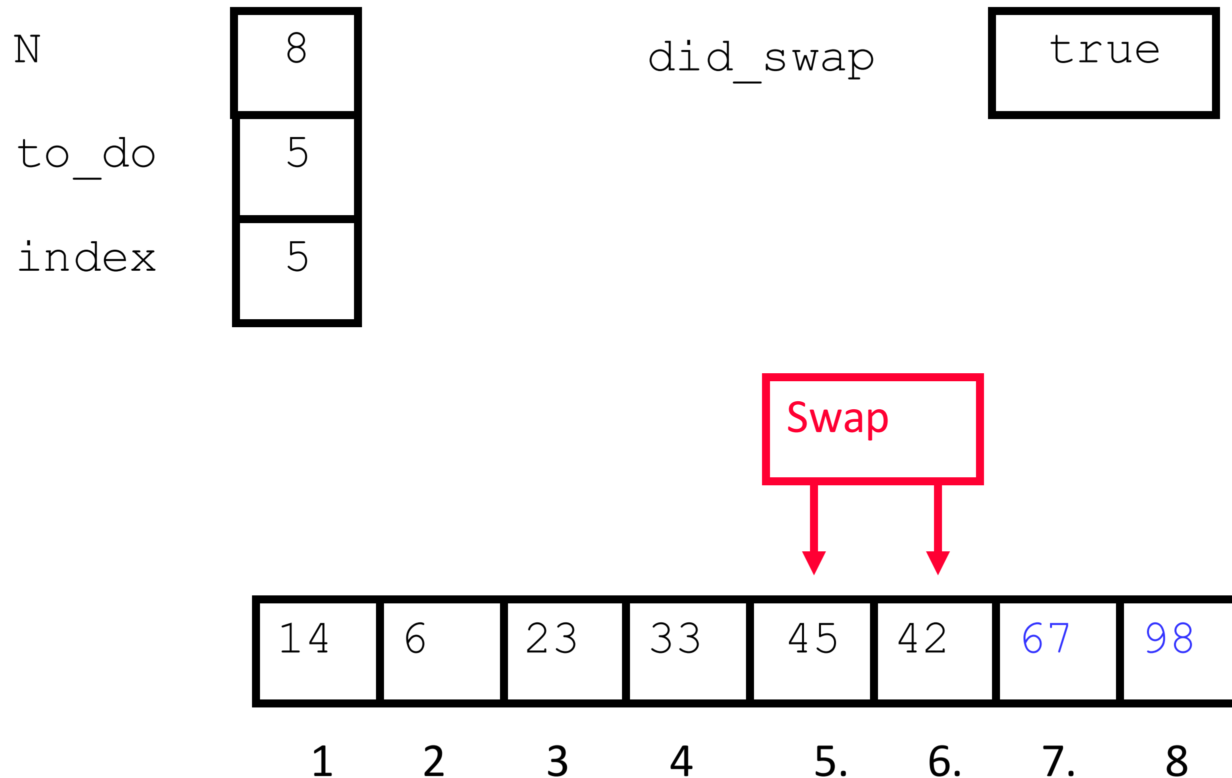


The Third "Bubble Up"

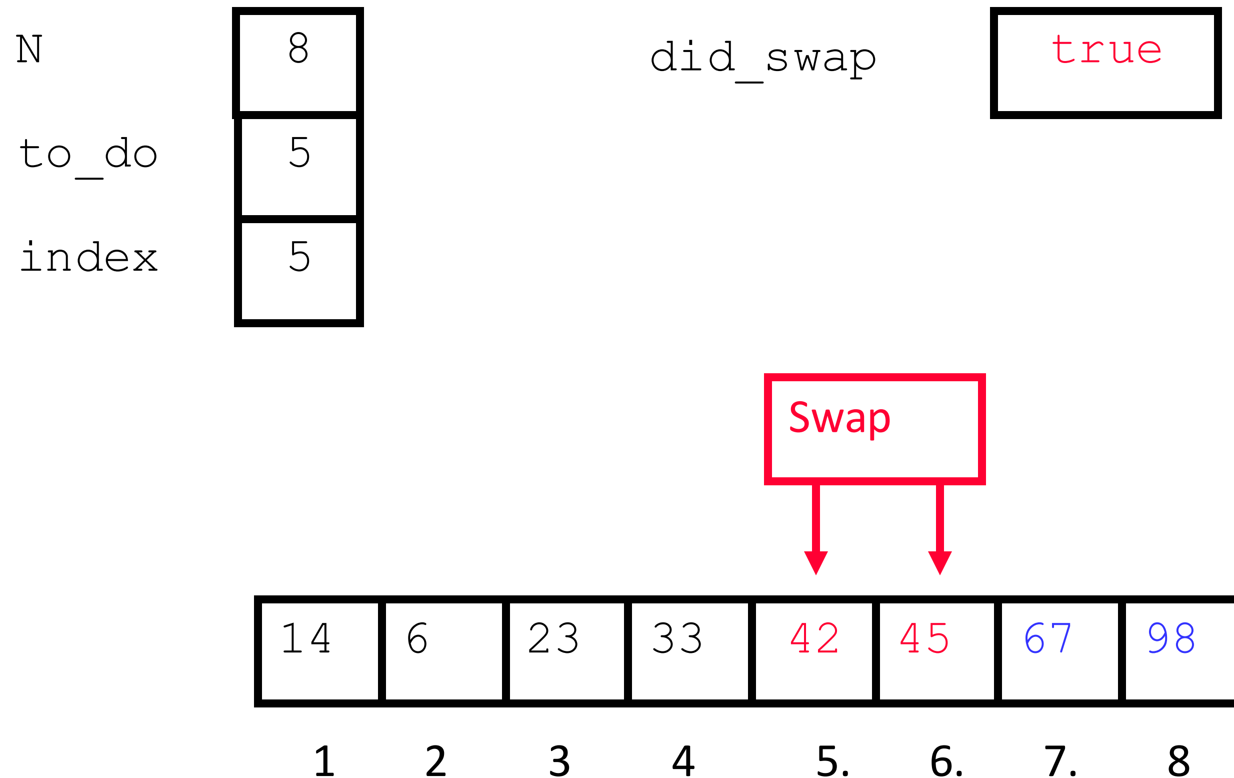




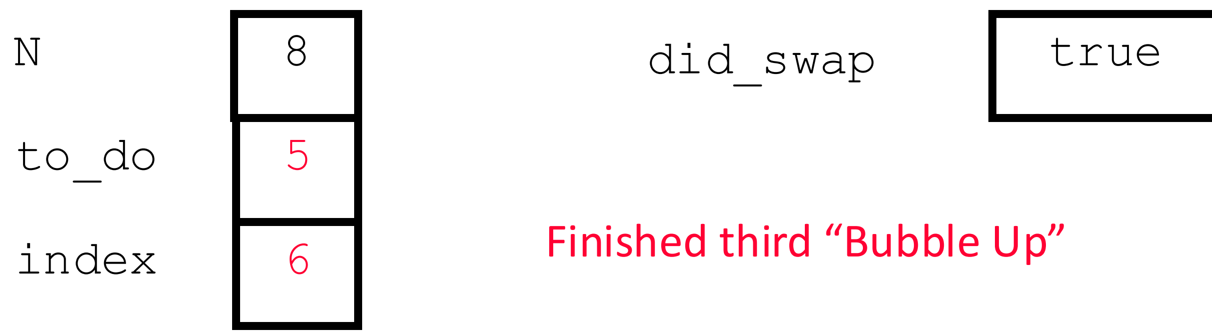
The Third “Bubble Up”

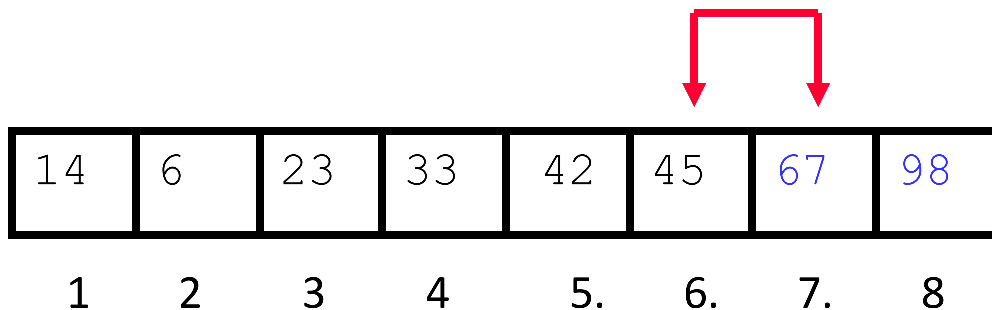


The Third "Bubble Up"

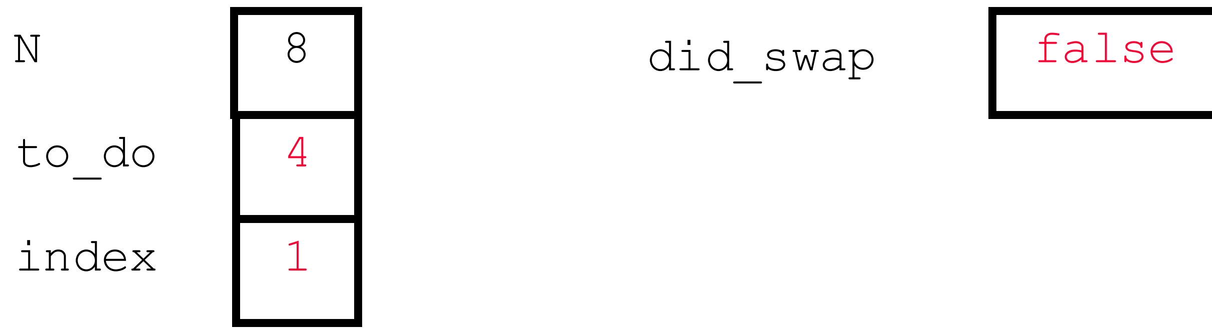


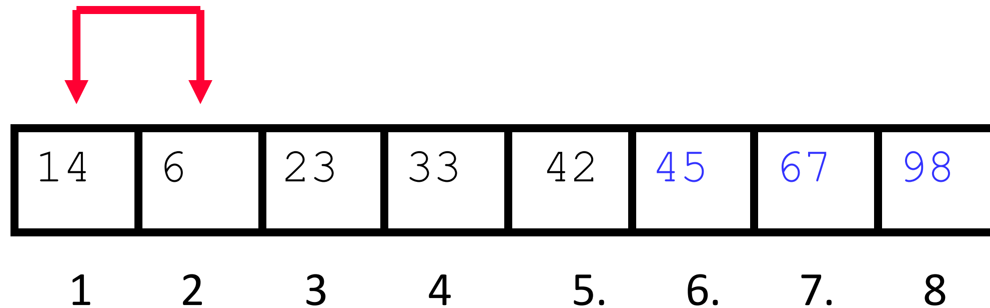
After Third Pass of Outer Loop



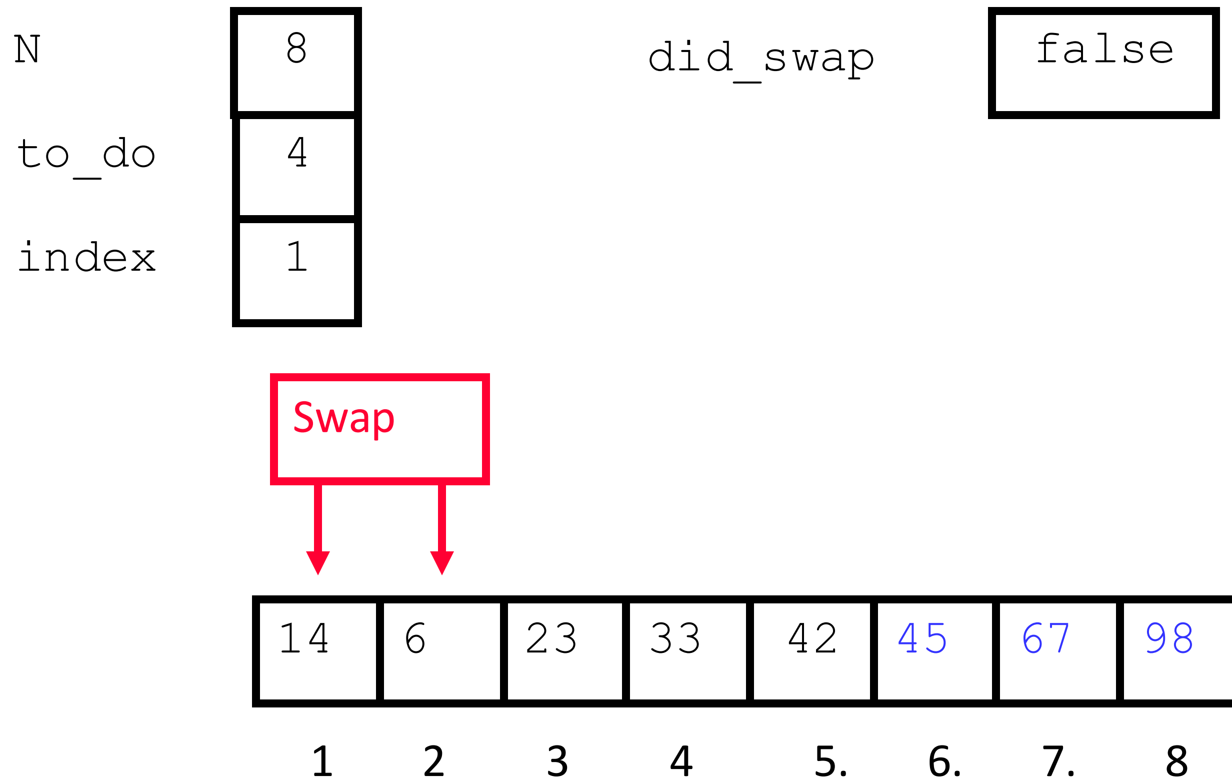


The Fourth "Bubble Up"

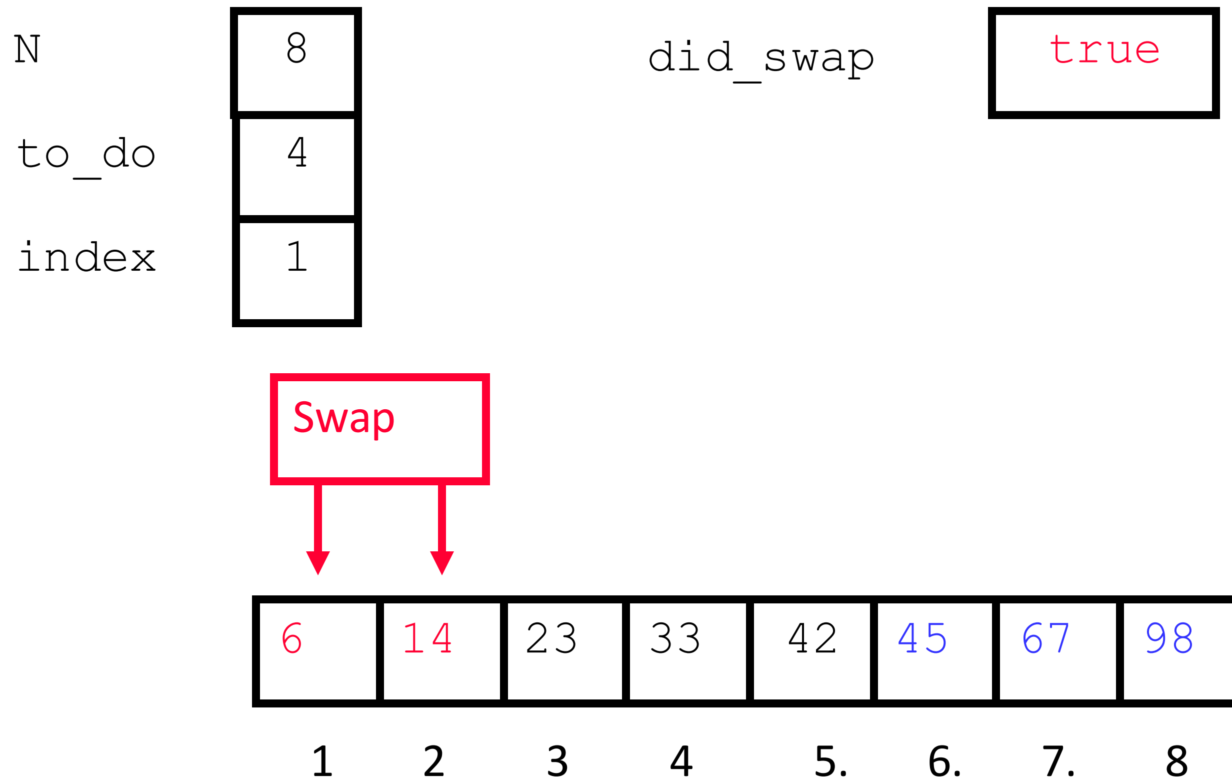




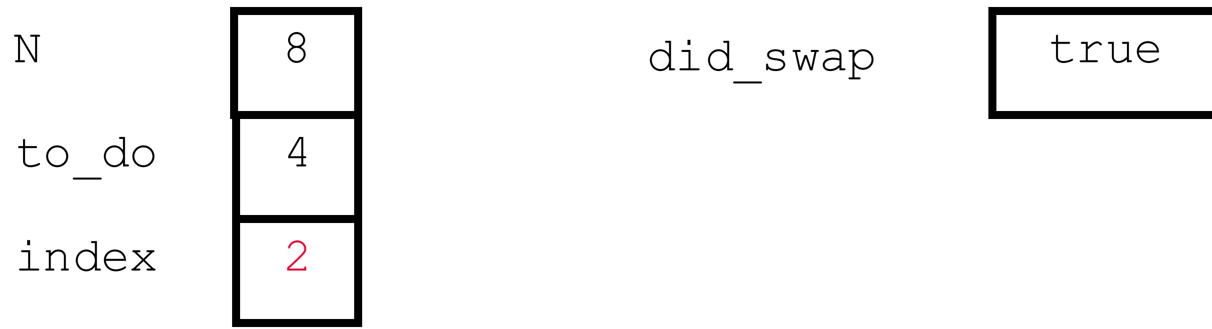
The Fourth “Bubble Up”

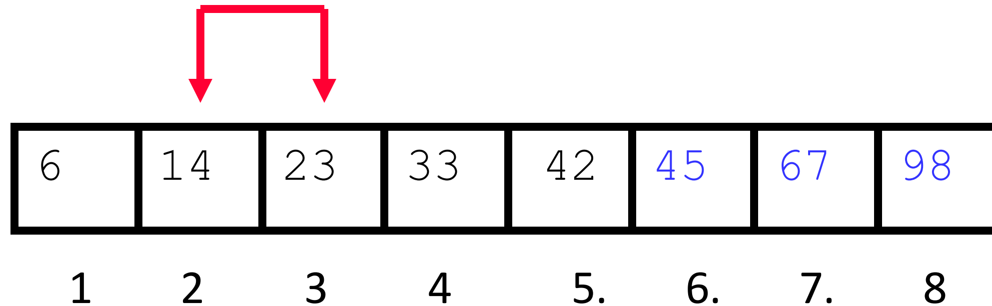


The Fourth "Bubble Up"

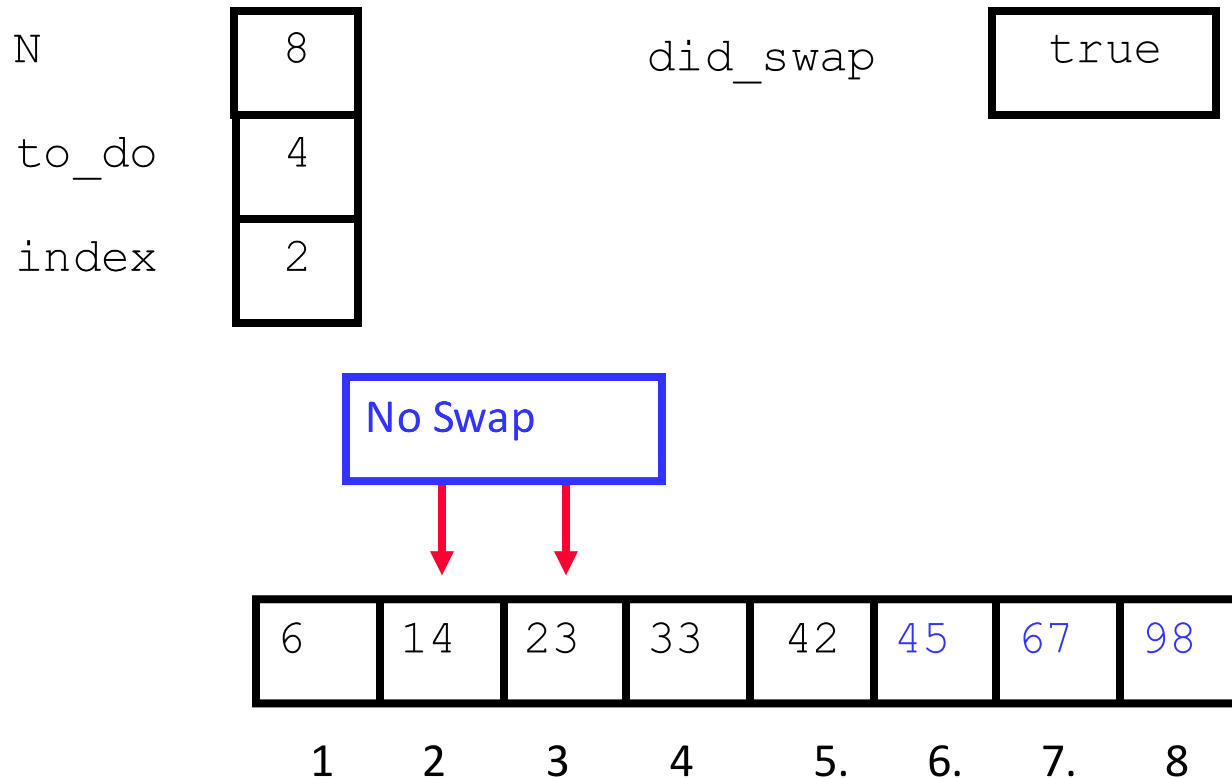


The Fourth "Bubble Up"

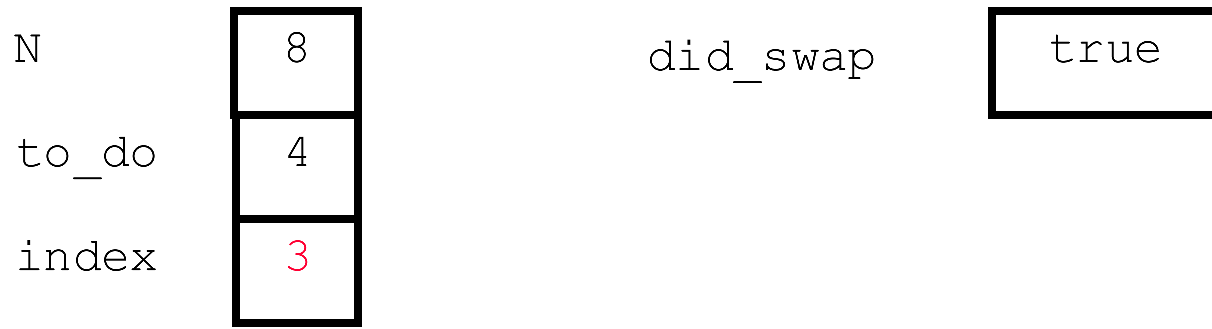


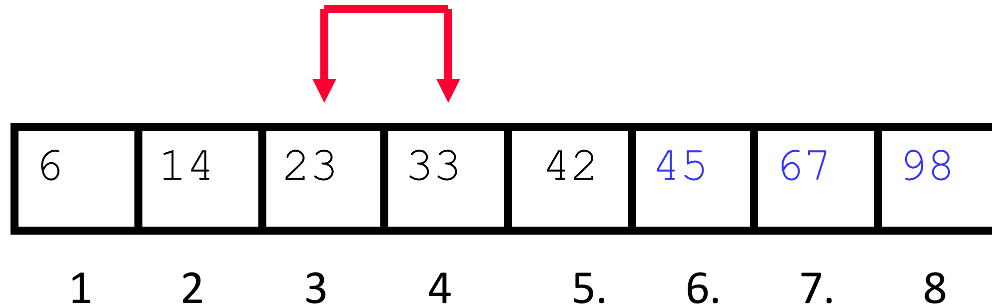


The Fourth "Bubble Up"

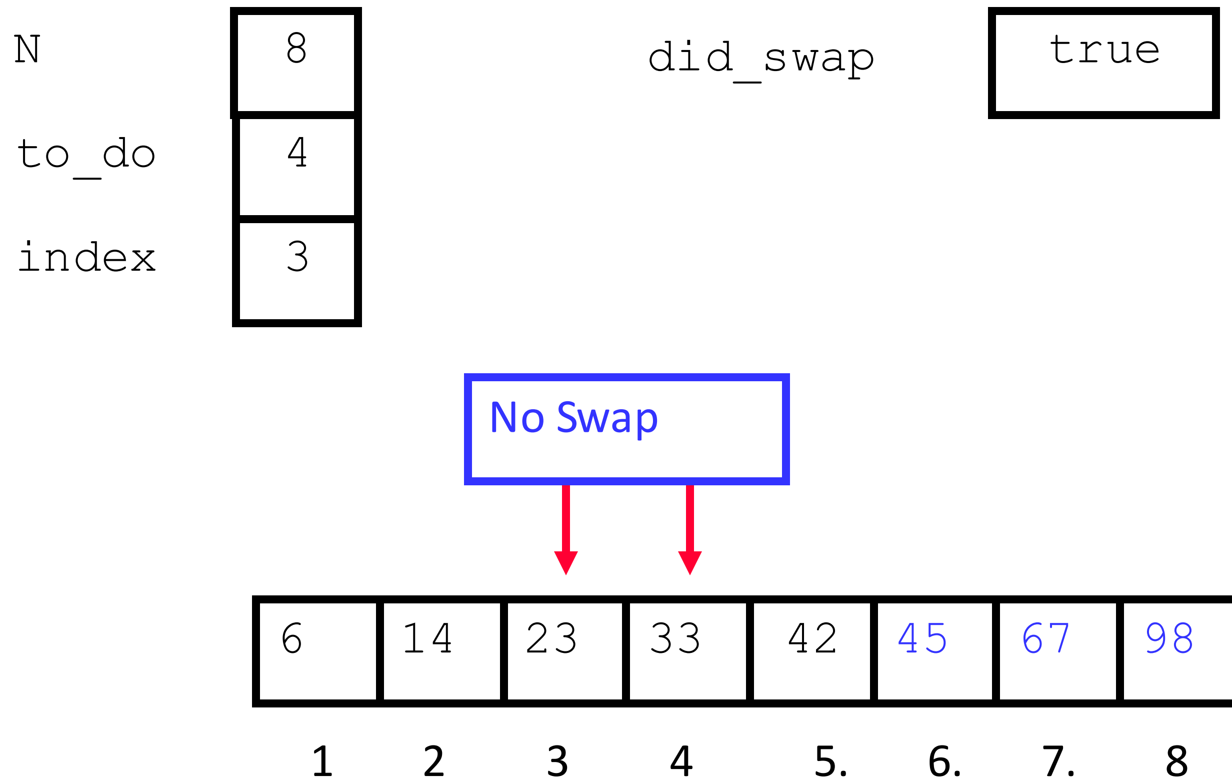


The Fourth "Bubble Up"

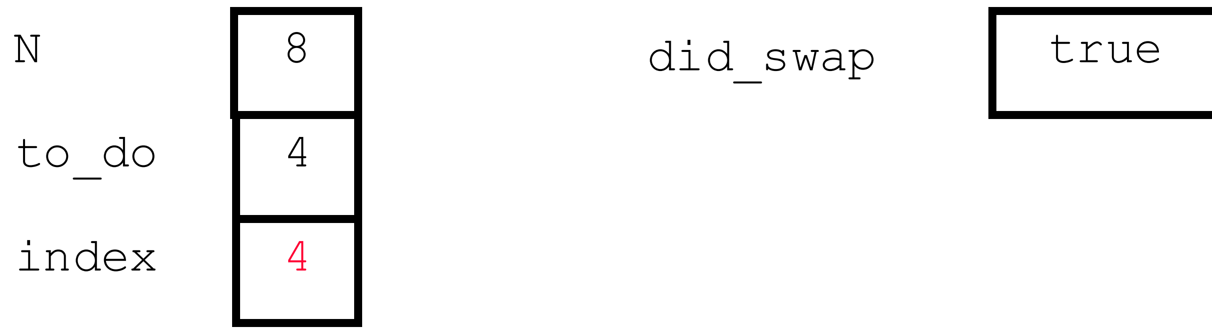


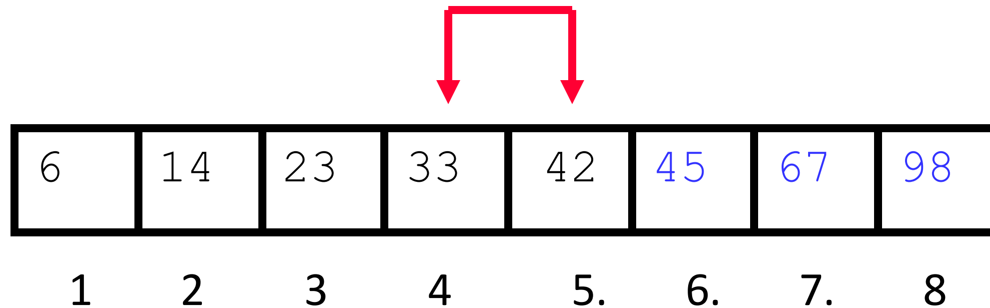


The Fourth "Bubble Up"

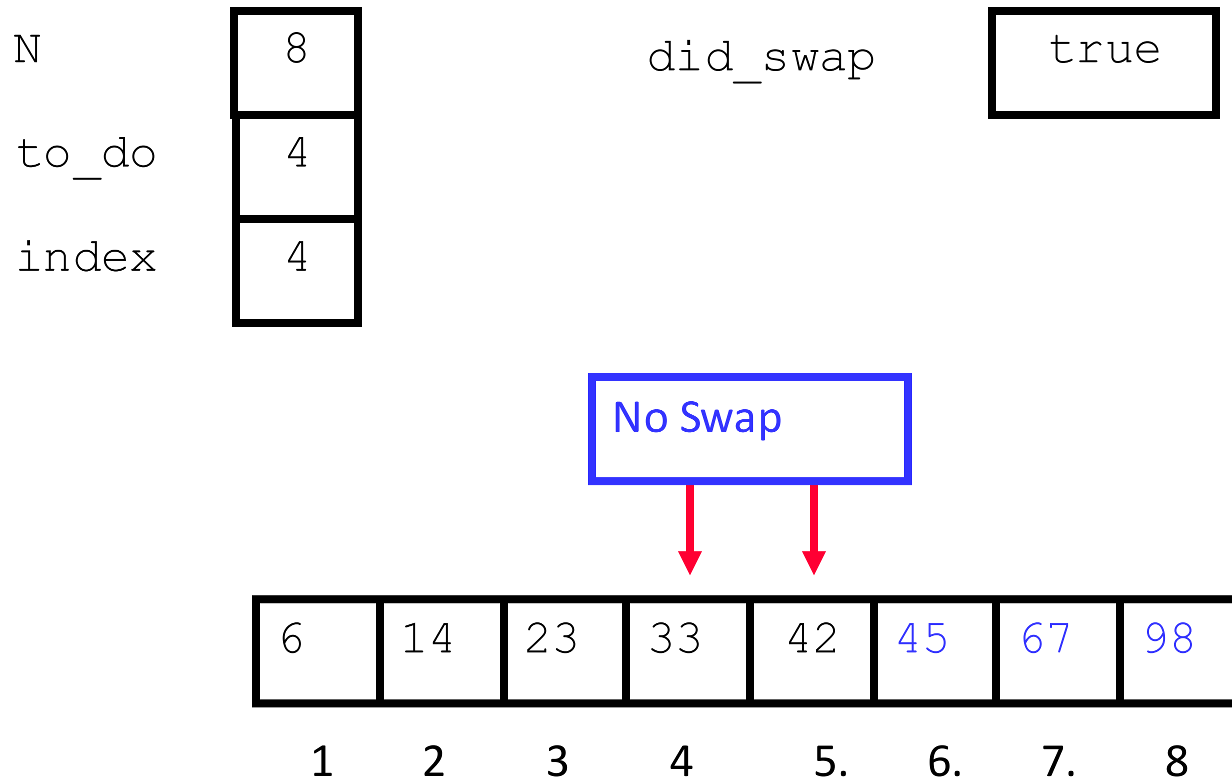


The Fourth "Bubble Up"

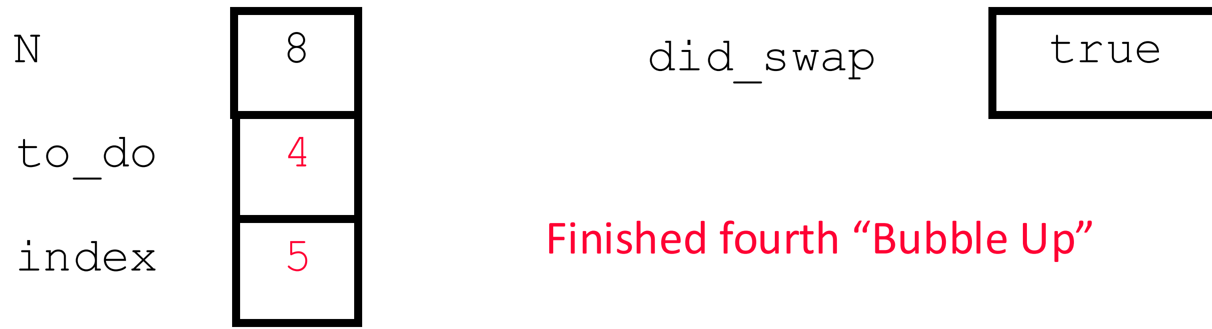


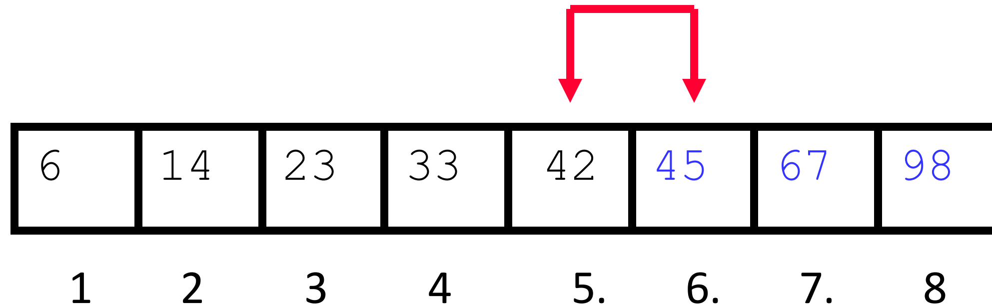


The Fourth "Bubble Up"

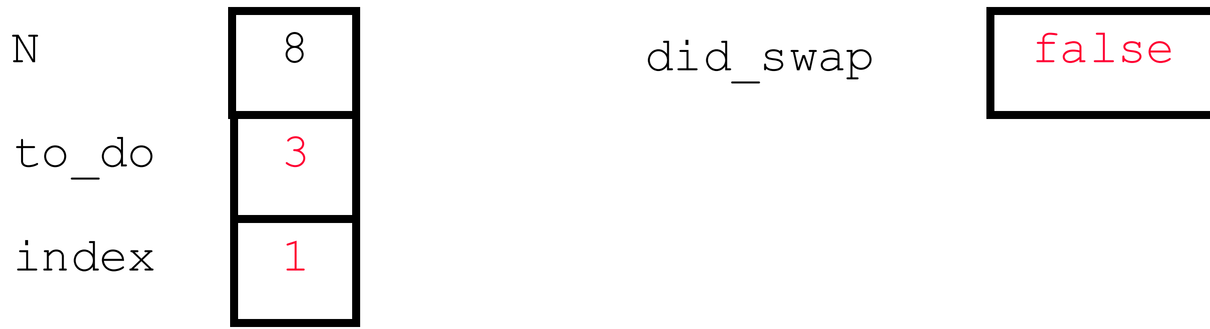


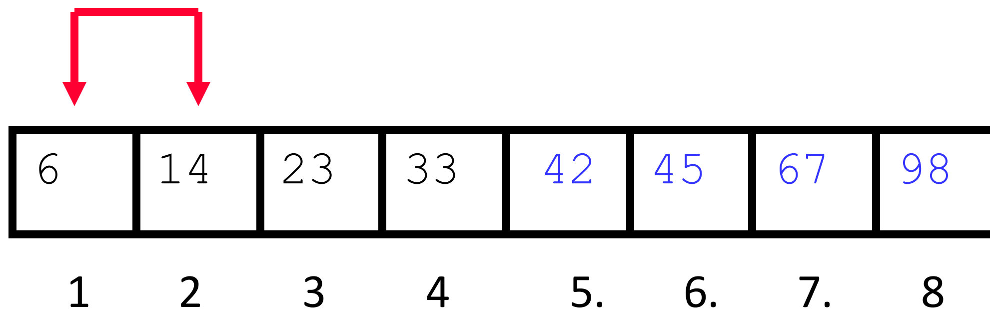
After Fourth Pass of Outer Loop



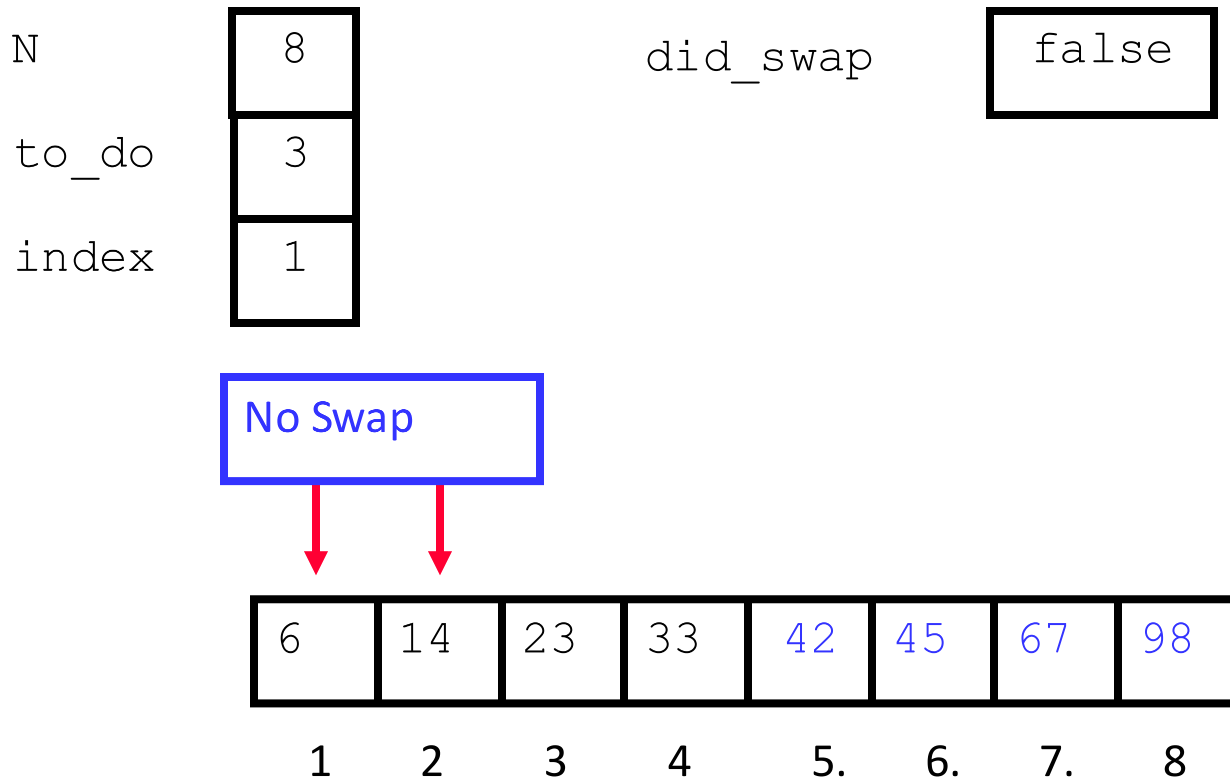


The Fifth "Bubble Up"

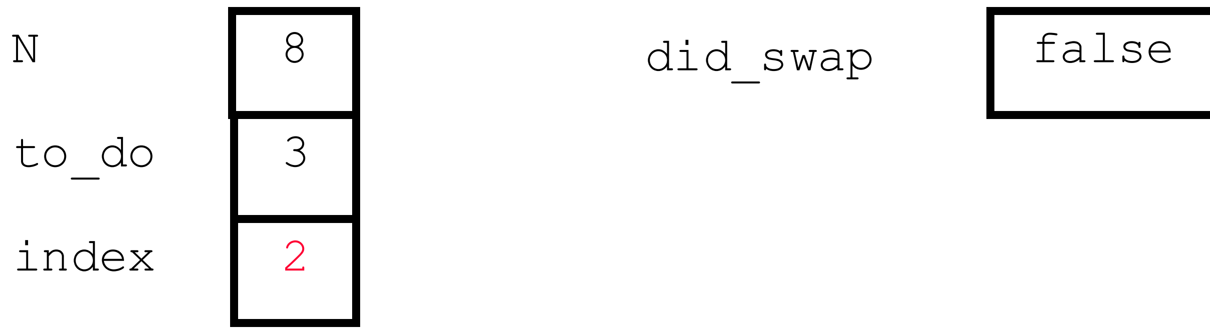


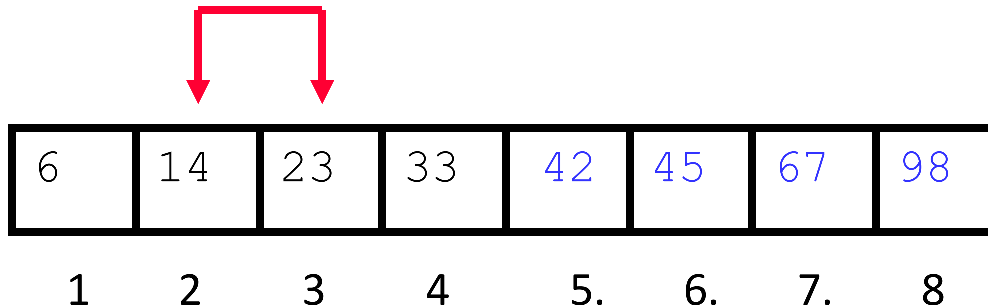


The Fifth "Bubble Up"

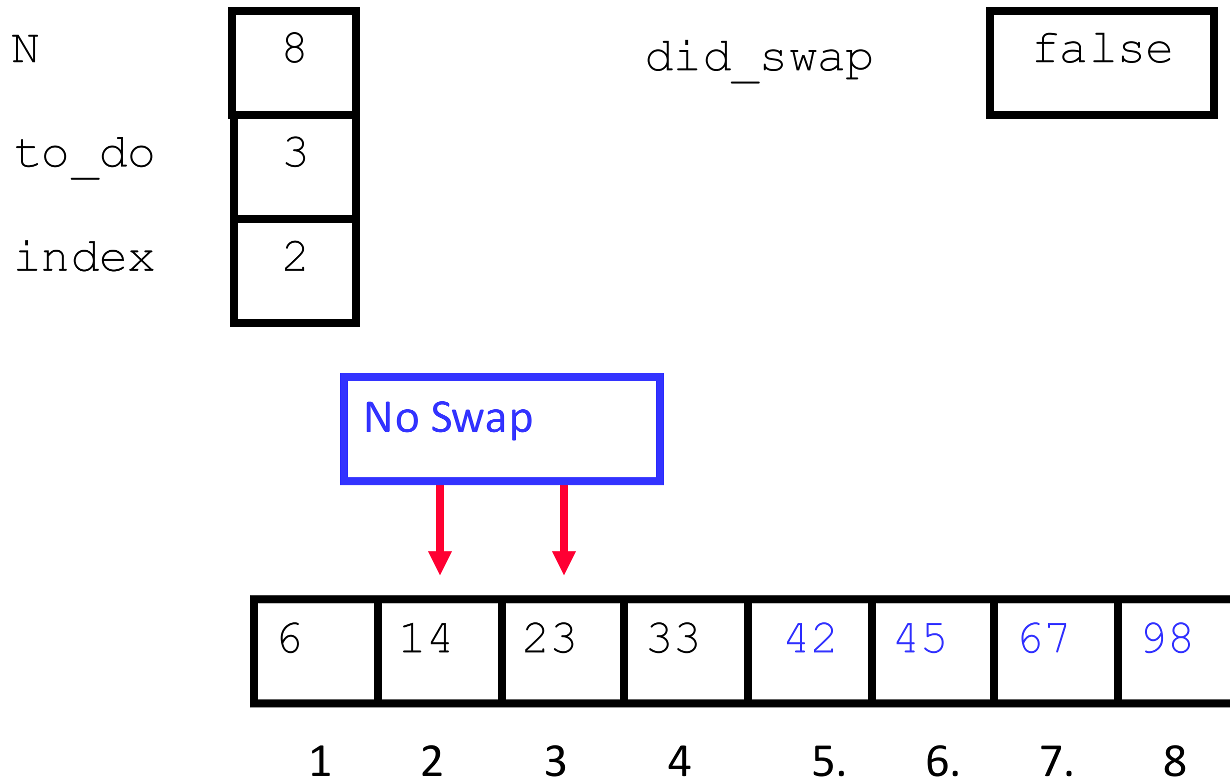


The Fifth "Bubble Up"

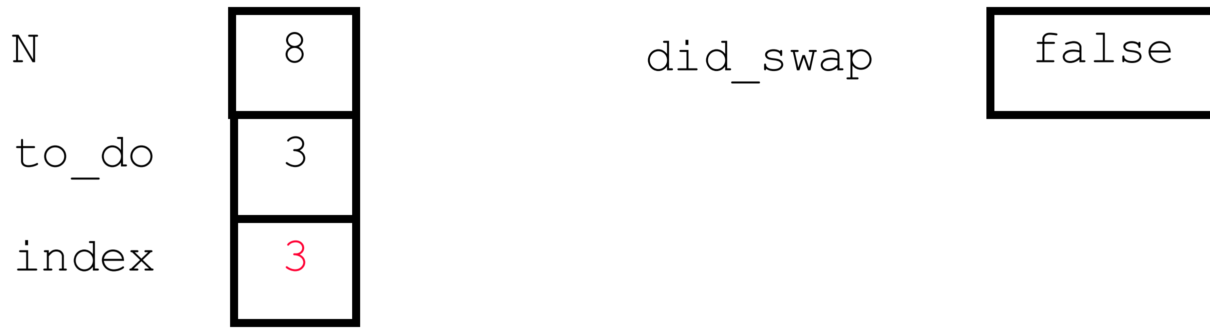


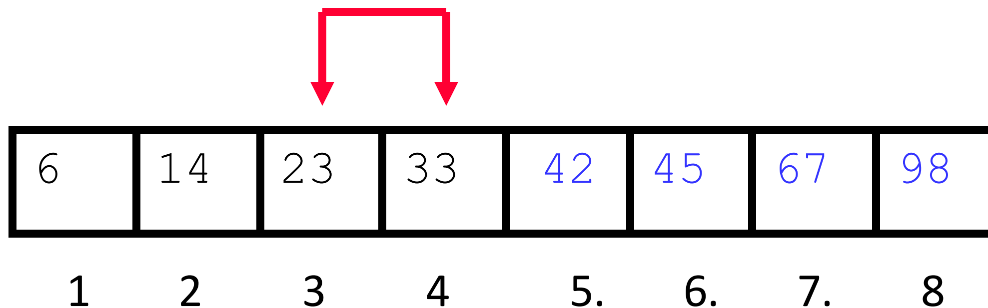


The Fifth "Bubble Up"

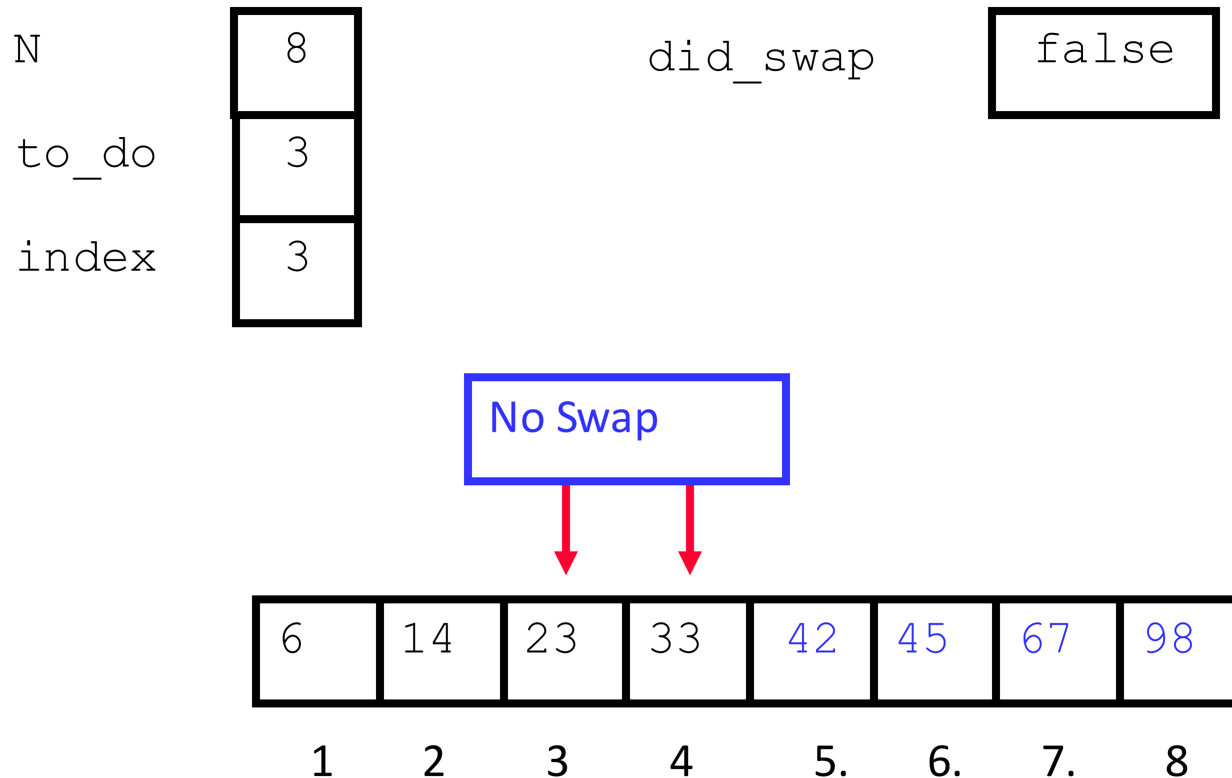


The Fifth "Bubble Up"

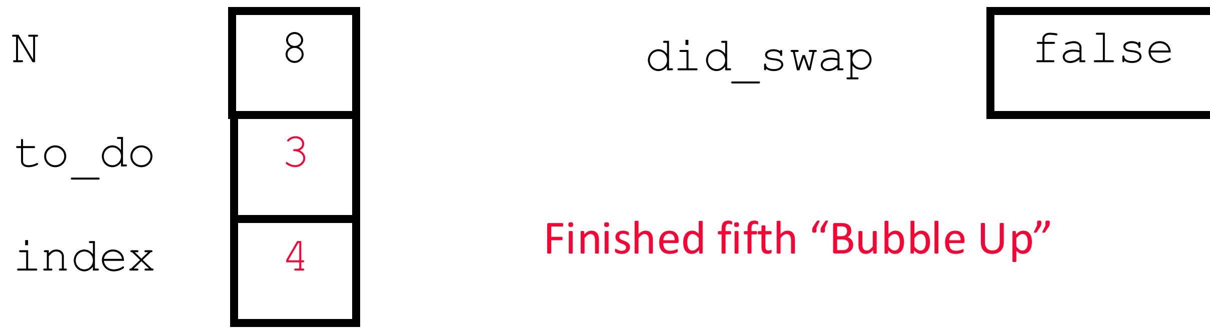


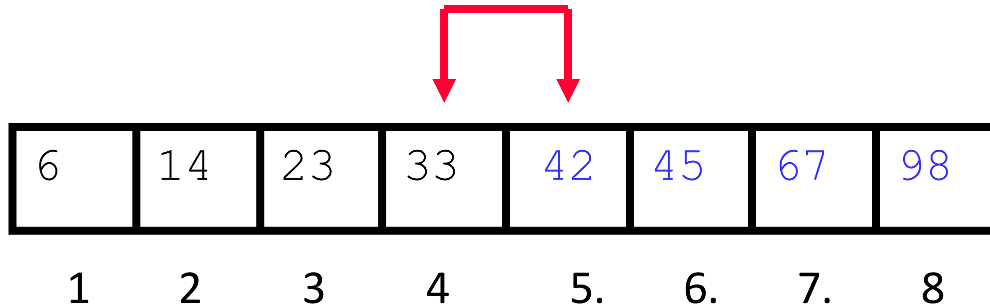


The Fifth "Bubble Up"

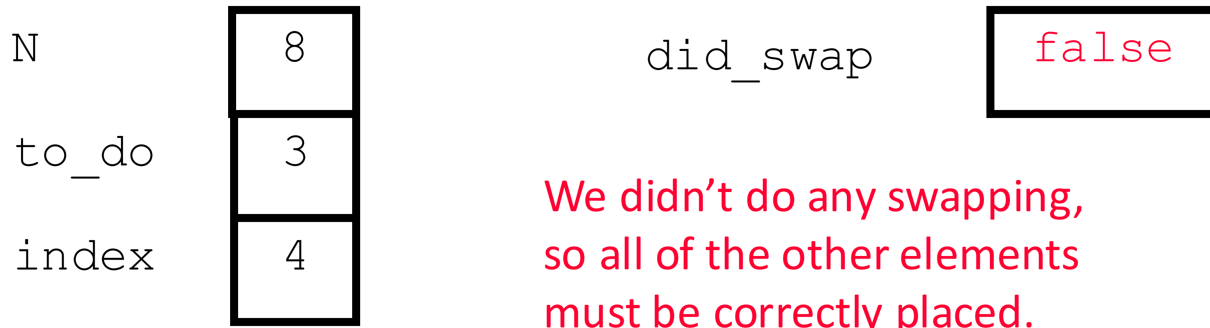


After Fifth Pass of Outer Loop



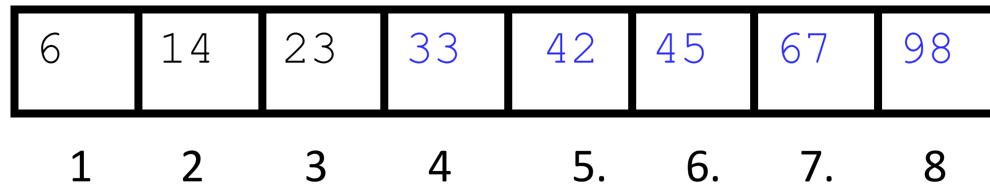


Finished "Early"



We didn't do any swapping, so all of the other elements must be correctly placed.

We can "skip" the last two passes of the outer loop.



Short bubble coding

```
def short_bubble_sort(a_list):
    exchanges = True
    pass_num = len(a_list) - 1
    while pass_num > 0 and exchanges:
        exchanges = False
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                exchanges = True
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp
```

Bubble Sort

- Complexity Analysis

Best case: O(n)

Worst case: $O(n^2)$

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
$\dots$	$\dots$
$n-1$	1

$(n-1) + (n-2) + \dots + 1 = n(n-1)/2 = (n^2-n)/2$

Summary

- "Bubble Up" algorithm will move largest value to its correct location (to the right)
- Repeat "Bubble Up" until all elements are correctly placed:
  - Maximum of N-1 times
  - Can finish early if no swapping occurs
- We reduce the number of elements we compare each time one is correctly placed
- Complexity is $O(n^2)$

Selection Sort

100

Selection Sort

- List sorted by selecting elements in the list
  - Select elements one at a time
  - Move elements to their proper positions
- Selection sort operation
  - Find location of the largest (smallest) element in unsorted list portion (starting from entire list)
  - Move it to bottom (top) of unsorted portion of the list
  - Continue with remaining unsorted list portion

101

Selection Sort

- Selection sort
  - Strategy
    - Select the largest item and put it in its correct place
    - Select the next largest item and put it in its correct place, etc.

Shaded elements are selected; boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

Figure 10-4

A selection sort of an array of five integers

Selection sort coding

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
$\dots$	$\dots$
$n-1$	1

Need a nested for loop structure

Need swap code: if a > b, swap a and b

```
if (a > b):
    temp = a
    a = b
    b = temp
```

```
if (a > b):
    a, b = b, a
```

Selection sort coding

Need a variable to save the position of the max

Selection sort coding

```
def selection_sort(a_list):
    for fill_slot in range(len(a_list) - 1, 0, -1): #index of the path
        pos_of_max = 0
        for location in range(1, fill_slot + 1): #index of the comparison
            if a_list[location] > a_list[pos_of_max]: #identify max location
                pos_of_max = location
        temp = a_list[fill_slot]  #swap
        a_list[fill_slot] = a_list[pos_of_max]
        a_list[pos_of_max] = temp


a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selection_sort(a_list)
print(a_list)
```

Selection Sort

- Analysis
  - Complexity is $O(n^2)$
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = (n^2-n)/2$
- Advantage of selection sort
  - The running time does not depend on the initial arrangement of the data (worst case running time is same as best case running time on all data sets)
  - Less swap, faster in practice than bubble up
- Disadvantage of selection sort
  - It is only appropriate for small n