List (Part I)

A linear collection of items

Assumptions

- The list is contiguous (“no gaps”).
- Items are identified by their position in the list.
- The first item is at position 1.

Primary methods (IList)

```
// Adds an item at the end of the list
public void Add(T item)

// Inserts an item at position p in the list
public void Insert(T item, int p)

// Removes the item at position p in the list
public void RemoveAt(int p)
```

Primary methods (IList)

```
// Returns true if item is removed from the list;
// false otherwise
public bool Remove(T item)

// Retrieves the item at position p
public T Retrieve(int p)

// Returns true if the item is found in the list;
// false otherwise
public bool Contains(T item)
```

Supporting methods (IContainer)

```
// Resets the list to empty
public void MakeEmpty()

// Returns true if the list is empty; false otherwise
public bool Empty()

Returns the number of items in the list
public int Size()

// Outputs the list to console (not part of IContainer)
public void Print()
```

Generic List class

```
class List<T> : IList<T>, IContainer<T>
{
    ...
}
```

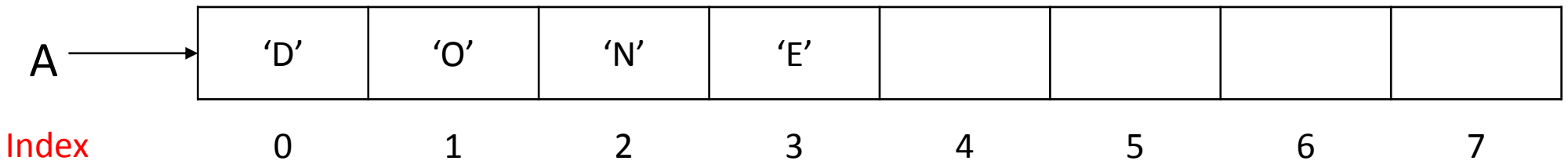
Note: In the code, IList inherits IContainer.

Data structures

- Linear array ←
- Singly linked list

Linear array

```
private T[] A;          // Linear array of items (Generic)
private int capacity;   // Maximum capacity of the list
private int count;      // Actual number of items in the list
                        // Note that position p is equivalent to index p-1
```

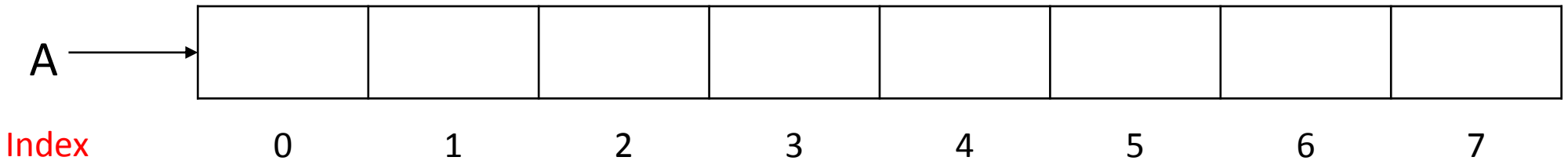


capacity = 8

count = 4

Constructor

- Basic Strategy
  - Create an array with a capacity of 8 (by default) and set count to 0.
  - A = new T [8] ;

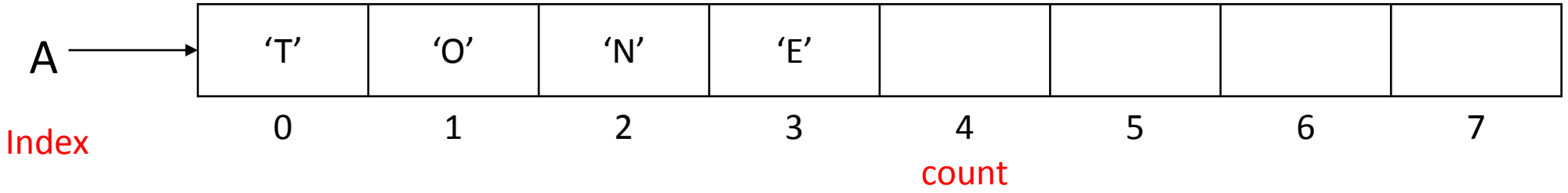


capacity = 8
count = 0

Add an item at the end of a list L

- Basic Strategy
  - If the list is full (count == capacity) then double the capacity of the list.
  - Place item at A[count] and increase count by 1.

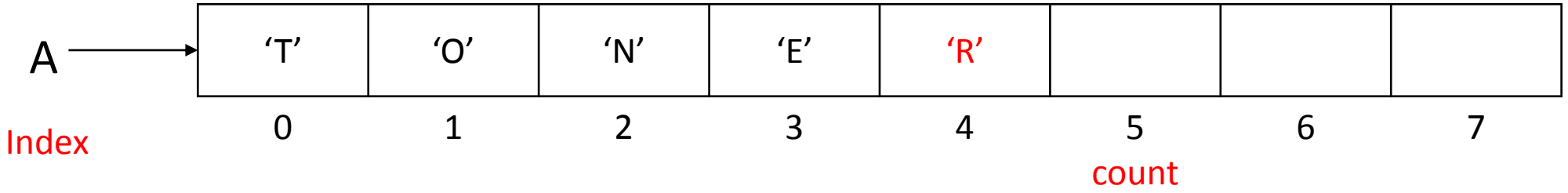
Add ‘R’ at the end of list L



capacity = 8

count = 4

Add ‘R’ at the end of list L



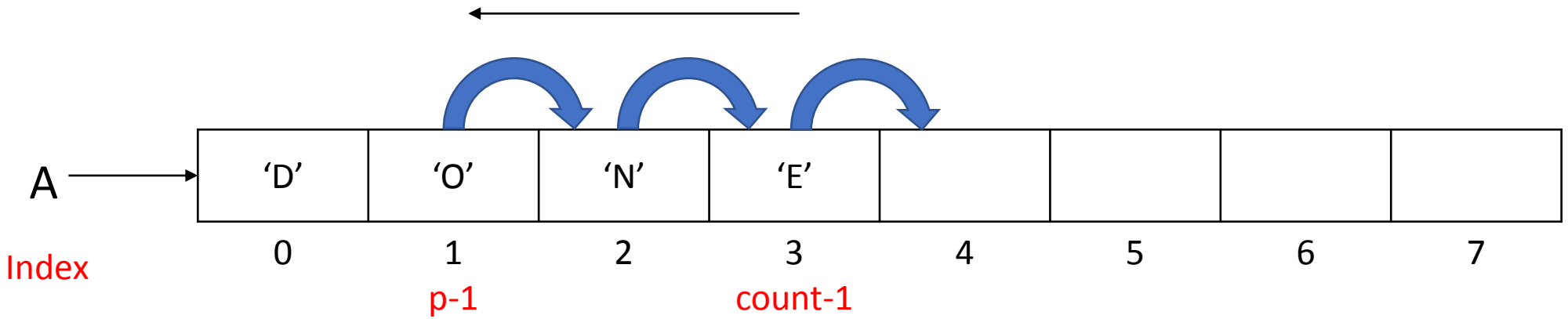
capacity = 8

count = 5

Insert item at position p in list L

- Basic Strategy
  - If position p is out-of-range (p < 1 || p > count+1), then throw an exception
  - Else
    - If the list is full, then double the capacity of the list.
    - Move up the current items in A from index count-1 down to index p-1.
    - Place item at A[p-1].
    - Increase count by 1.

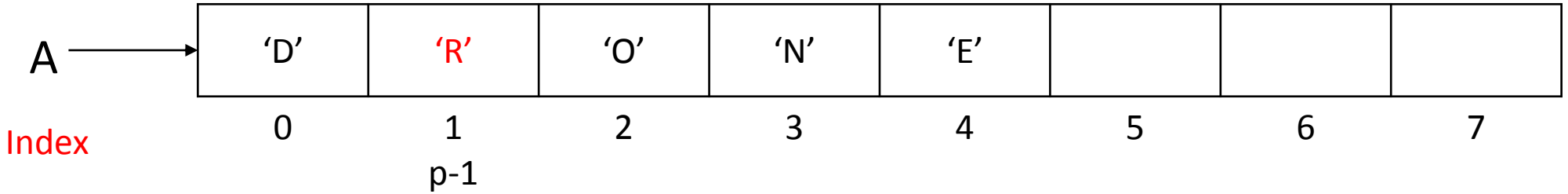
Insert ‘R’ at position p=2 in list L



capacity = 8

count = 4

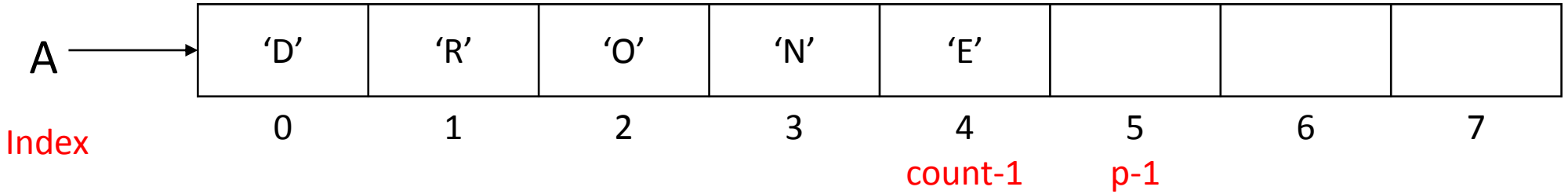
Insert ‘R’ at position p=2 in list L



capacity = 8

count = 5

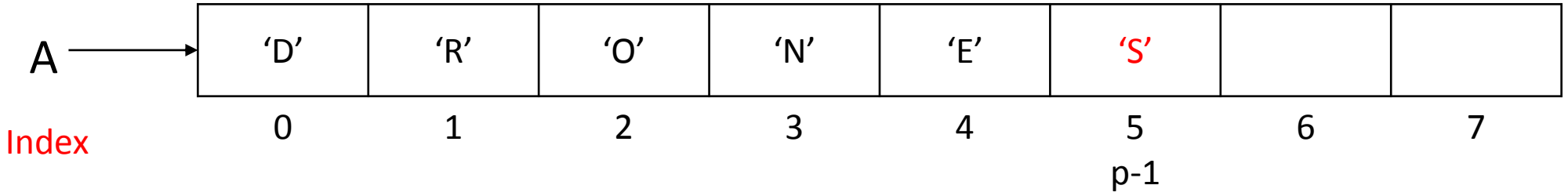
Insert ‘S’ at position p=6 in list L



capacity = 8

count = 5

Insert ‘S’ at position p=6 in list L



capacity = 8

count = 6

Digression on the Big-Oh notation

- What is the minimum and maximum number of items that must be moved to insert an additional item into a list with n items?
  - The minimum is 0 items when the additional item is inserted at the end of a list.
  - The maximum is n items when the additional item is inserted at the front of the list.

- In the worst case, approximately n total steps are required to insert an additional item into a list with n items.

- To succinctly state the relationship between the size n of a problem and the total number of steps to the solve the problem, the Big-Oh notation is used.

- In the case of the Insert method, this relationship is linear and is stated using the Big-Oh notation as:

O(n)

- If the size of a problem has no bearing on the total number of steps to solve the problem, then the relationship is constant and is stated using the Big-Oh notation as:

$O(1)$

- If the relationship is quadratic, the Big-Oh notation is:

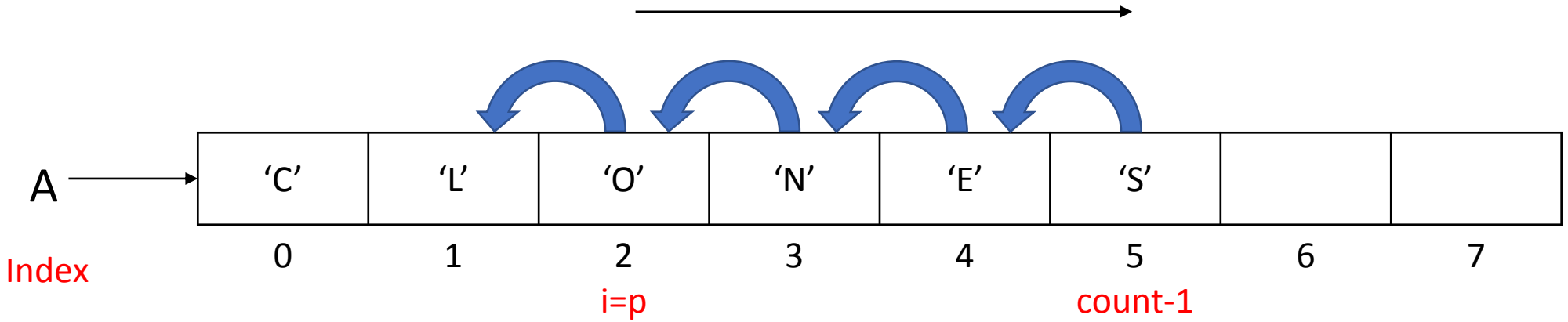
$O(n^2)$

- Since the total number of steps to solve a problem is a good proxy for the time it takes to solve the problem, the Big-Oh notation is therefore used to:
  - express the time complexity of an algorithm and
  - succinctly show the relationship between problem size n and time T(n).

Remove item at position p in list L

- Basic Strategy
  - If the list is empty or position is out-of-range, then throw an exception.
  - Else
    - Move down the current items in A from index p up to count-1.
    - Decrease count by 1.

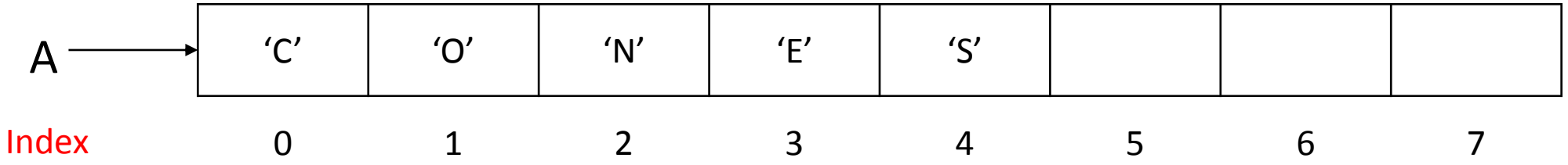
Remove item at position p=2 in list L



capacity = 8

count = 6

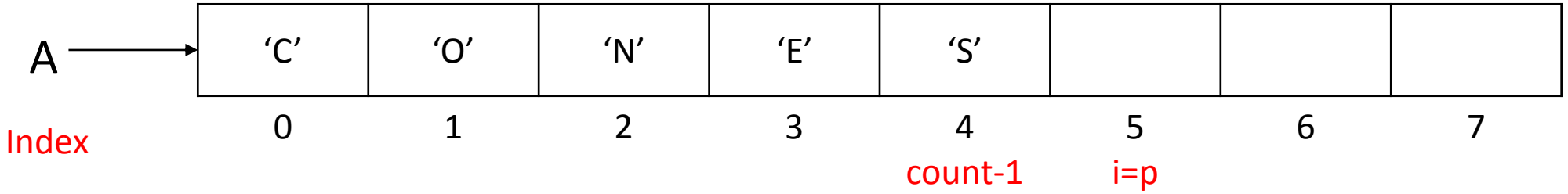
Remove item at position p=2 in list L



capacity = 8

count = 5

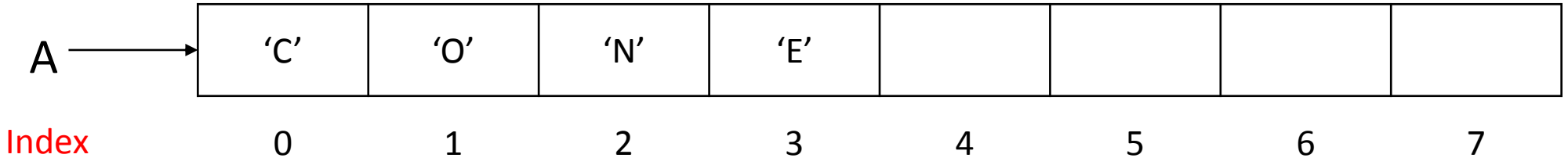
Remove item at position p=5 in list L



capacity = 8

count = 5

Remove item at position p=5 in list L



capacity = 8

count = 4

Worst-case time complexity of RemoveAt

- To remove the item in the first position of a List, all other n-1 items must be moved down. The worst-case time complexity of Remove is therefore:

O(n)

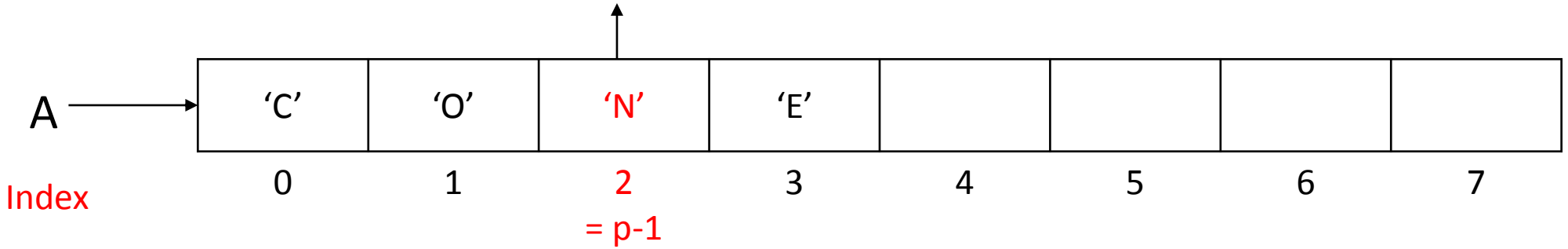
Remove item from list L

- Basic Strategy
    - If the list is empty, then throw an exception.
    - Else
        - Search for the item in the list.
        - If not found, then return false.
        - If found
            - Call RemoveAt(i) where i is the index in A of the first instance of item.
            - Return true.

Retrieve item at position p in list L

- Basic Strategy
    - If position is out-of-range, then throw an exception.
    - Else
        - Return item at index p-1 where 1 <= p <= count.

Retrieve item at position p=3 in list L



capacity = 8

count = 4

Worst-case time complexity of Retrieve

- Since the time required to retrieve an item from a List does not depend on the size of the List, the worst-case time complexity of Contains is:

O(1)

Contains: Does list L contain item?

- Basic Strategy
  - Traverse A until either the list is fully examined, or the item is found.
  - Return true if the item is found; false otherwise.

```
public bool Contains(T item)
{
    int i = 0;
    bool found = false;

    while (i < count && !found)
        if (A[i].Equals(item))
            found = true;
        else
            i++;
    return found;
}
```

Worst-case time complexity of Contains

- If the item is not found in the list, then the entire list is traversed. Because all n items are examined, the worst-case time complexity of Contains is:

O(n)

Supporting methods

- MakeEmpty
  - Sets count to 0 (only)
  - O(1)
- Empty
  - Returns true if count is 0; false otherwise
  - O(1)
- Size
  - Returns count
  - O(1)

Exercises

1. What happens when the for loop is reversed in the Insert method?

2. What happens when the for loop is reversed in the RemoveAt method?

3. MakeEmpty resets count to 0, but all items remain in the array. Why does this not matter?

Exercises (con’t)

4. What sequence of p values would lead to the minimum and maximum number of total steps to remove all items from an initial list with n items using the RemoveAt method? State the worst-case time complexity to remove all items using the Big-Oh notation.