Homework Sheet 4

Author	Matriculation Number	Tutor
Abdullah Oğuz Topçuoğlu	7063561	Maryna Dernovaia
Ahmed Waleed Ahmed Badawy Shora	7069708	Jan-Hendrik Gindorf
Yousef Mostafa Farouk Farag	7073030	Thorben Johr

Exercise 2

(a)

We will have the Stack structure like this

```
struct Stack {
  int top; // index of the top element
  int A[N]; // array of size N
};
```

So removing k many elements from stack can be done by just decrementing the top index by k. Here is the pseudocode:

```
function ManyPop(Stack S, int k)
  if S.top < k then
    S.top = 0; // stack becomes empty
  else
    S.top -= k; // remove top k elements
```

This implementation runs in $O(1)$ time since it only involves a couple of arithmetic operations and a conditional check.

(b)

We will have the Stack structure like this

```
struct Node {
  int data;
  Node* next;
};

struct Stack {
  Node* top; // pointer to the top node
};
```

Every element on the stack points to the elements thats below it. So the first element thats pushed to the stack has its next pointer as null. Here is the pseudocode for ManyPop:

```
function ManyPop(Stack S, int k)
  count = 0;
  while S.top != null and count < k do
    temp = S.top;
    S.top = S.top.next; // move top to the next element
    delete temp; // free memory of the popped element
    count += 1;
```

This implementation runs in $O(k)$ time since it involves a loop that iterates k times, performing constant time operations in each iteration.

(c)

To prove that the amortized running time of Push and ManyPop is $O(1)$, we will use the bank accounting method from the lecture. We will assign an amortized cost to each operation as follows:

- Push operation: Lets say we get 2 euros every time Push() is called. We spend 1 euro for the constant time operation in Push() function and store the remaining 1 euro in the bank.
- ManyPop operation: Lets say we get 0 euro every time ManyPop() is called. We need k euros to remove k elements from the stack. Meaning that we need 1 euro to remove one element. Luckily enough we saved 1 euro per element during the Push() operations so we can spend them here.

Exercise 3

This is not a great idea because when we shrink when $n = N/2 - 1$ the new size would be $2n = N - 2$. So we are just reducing the size by 2. And in the lecture we saw that adding a constant amount when resizing(growing) is a bad idea so we can guess that something similar would happen when removing(shrinking) a constant amount too.
Consider this sequence

```
// we start with a single push back
PushBack(x1) // array count becomes 1 // capacity here is 1

// we continue with two push backs and then two pop backs and repeat this as many times as y
PushBack(x2) // array count becomes 2 // capacity here is 2
PushBack(x3) // array count becomes 3 // capacity here is 4
PopBack()    // array count becomes 2
PopBack()    // array count becomes 1 // resizing happens here, new capacity becomes 2

PushBack(x4) // array count becomes 2
PushBack(x5) // array count becomes 3 // resizing happens here, new capacity becomes 4
PopBack()    // array count becomes 2
```

```
PopBack()    // array count becomes 1 // resizing happens here, new capacity becomes 2
```

So every second operation we have a resizing and resizing is linear time. So if we have m operations we would have $\frac{m}{2}$ resizings and each resizing takes linear time. So the total time would be $O(m^2)$.