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)

The initial state: $A_0 = [], A_1 = [], A_2 = []$
After insert(2): $A_0 = [2], A_1 = [], A_2 = []$
After insert(4): $A_0 = [], A_1 = [2, 4], A_2 = []$
After insert(10): $A_0 = [10], A_1 = [2, 4], A_2 = []$
After insert(5): $A_0 = [], A_1 = [], A_2 = [2, 4, 5, 10]$
After insert(6): $A_0 = [6], A_1 = [], A_2 = [2, 4, 5, 10]$
After insert(7): $A_0 = [], A_1 = [6, 7], A_2 = [2, 4, 5, 10]$

(b)

Pseudocode:

```
1: function FIND(k)
2:     for i := 0, 1, ... do                    ▷ iterate through each array
3:         x = A_i.BinarySearch(k)              ▷ Binary search in sorted array A_i
4:         if x ≠ ⊥ then
5:             return x
6:         end if
7:     end for
8:     return ⊥
9: end function
```

Correctness proof:

- Each array A_i is sorted. By the correctness of the binary search algorithm (from the lecture), A_i.BinarySearch(k) correctly returns the element of key k if it exists in A_i, and $\perp$ otherwise. - Since the algorithm iterates over all arrays, it will find k in the first array that contains it. - If k is not present in any array, the algorithm correctly returns $\perp$.

Hence, the algorithm correctly implements FIND.

Time analysis:

- Each array A_i has size $|A_i| = 2^i$, for $i = 0, 1, \ldots, \lceil \log n \rceil$. - Binary search on array A_i takes $O(\log |A_i|) = O(\log 2^i) = O(i)$ time. - In the worst case, we may search all arrays, so the total time is:

$$T_{\text{worst}} = \sum_{i=0}^{\lceil \log n \rceil} O(i) = O\Big(\sum_{i=0}^{\lceil \log n \rceil} i \Big) = O\big((\log n)^2\big)$$

- Therefore, the worst-case running time of FIND is $O((\log n)^2)$.

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 $O(m^2)$.

Exercise 4

- 1: **function** REMOVEDUPLICATE($A[1..n]$)
  2: Allocate $B[1..m]$
  3: $C :=$ new Dictionary
  4: $j := 1$
  5: **for** $i := 1$ to n **do** ▷ $O(n)$
  6: **if** C.find$(A[i]) = \perp$ **then**
  7: C.insert$(A[i])$ ▷ $O(\log n)$
  8: $B[j] := A[i]$
  9: $j = j + 1$
  10: **end if**
  11: **end for**
  12: **return** B
  13: **end function**

Correctness Proof:

We show that the algorithm REMOVEDUPLICATE correctly returns an array B that contains all elements of A in their original order, with duplicates removed.

Case 1: A has no duplicate elements.

- For each i from 1 to n, the lookup C.find$(A[i])$ will return $\perp$ because the element $A[i]$ has not been inserted before.
- Therefore, the condition `if C.find(A[i]) =` $\perp$ is always true, so every element $A[i]$ is inserted into both the dictionary C and the output array B.
- Consequently, B will contain exactly all elements of A in the same order, and no elements are skipped.

Hence, if A has no duplicates, $B = A$.

Case 2: A contains duplicates.

- Consider the first occurrence of any value x in A. At this point, C.find$(x) = \perp$, so x is inserted into C and appended to B.
- For any subsequent occurrence of the same value x, we have C.find$(x) \neq \perp$, so the algorithm skips the insertion and does not append x to B.

– Thus, each distinct element of A appears exactly once in B, and since we traverse A in order, the relative order of the first appearances is preserved.

Conclusion: In both cases, B contains exactly the distinct elements of A, preserving their order of first appearance. Therefore, the algorithm is *correct*.

Time analysis: Each of the n iterations performs one dictionary lookup and possibly one insertion, each taking $O(\log n)$ time, so the total running time is $O(n \log n)$.

- **1. Binary Search Tree (BST):**

  Let the BST have n elements and height h. In a basic (unbalanced) BST, h can be as large as n. - Insert(x) requires descending from the root to the appropriate leaf: $\Theta(h)$ time. - Remove(x) also requires searching for x and possibly adjusting the tree: $\Theta(h)$. - Min() requires following the leftmost path: $\Theta(h)$.

  Even in a balanced BST, $h = \Theta(\log n)$. Therefore, at least one of the operations Insert, Remove, or Min requires $\Omega(\log n)$ time.

  2. Sorted Array :

  In a sorted array of size n: - Min() can be performed in $\Theta(1)$ time (the first element). - Insert(x) can find the correct position using binary search in $\Theta(\log n)$ time, but it must shift all subsequent elements to make space: $\Theta(n)$ worst case. - Remove(x) can locate the element using binary search in $\Theta(\log n)$ time, but shifting elements after removal costs $\Theta(n)$ in the worst case.

  Hence, at least one operation (Insert or Remove) requires $\Omega(n)$ time, which certainly implies $\Omega(\log n)$.

  3. Unsorted Array:

  In an unsorted array: - Insert(x) can be done in $\Theta(1)$ time (append at the end). - Remove(x) requires locating x by scanning the array: $\Theta(n)$. - Min() requires scanning all n elements: $\Theta(n)$.

  Thus, at least one operation (Remove or Min) requires $\Omega(\log n)$ time.

  4. Linked List:

  For a singly linked list: - Insert(x) at the head or tail can be done in $\Theta(1)$ time. - Remove(x) requires scanning for x: $\Theta(n)$. - Min() requires traversing all n elements: $\Theta(n)$.

  Again, at least one operation requires $\Omega(\log n)$ time.