CS 225: Data Structures

Homework 2

Group D1

Last Modified on March 4, 2022

Members	
Name	**Student ID**
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

Please turn over for our answers.

Ex. 1 **Our Answer.**

We define two operations: checking-in (happens at arrival date) and checking-out (happens at departure date). Let the priority of checking-out is higher than checking-in, if they happen on the same day. Then one single booking will contain a checking-in and a checking-out.

In order to maximize the use rate of hotel rooms, we assume the hotel only handle with check-ins in the afternoon and departures in the morning (i.e. A departure always happens earlier than an arrival, if they are on the same date).

Algorithm 1: Hotel Room Check

```
/* NOTE: Sort is a modified version of QuickSort, sorting the operation with the two
   criteria: 1. from earlier date to later; 2. on the same day, arrivals are put after
   departures. */
/* It outputs a list of operations with a member prop indicating whether the operation
   is arrival or departure. This tagging process is also completed in Sort. */
if k ≤ 0 then
    print("Bad Hotel!!!!!! GIVE MY MONEY BACK!!!!!!")
end
sortedOps ← Sort(bookings.arrivalDate, bookings.departureDate)
foreach operation(i) in sortedOps do
    if operation(i).prop = arrival then
        if numInUse = k then
            print("Insufficient Rooms!")
            Exit Algorithm
        end
        else
            numInUse = numInUse + 1
        end
    end
    else
        if numInUse = 0 then
            print("Bad testing engineer! I will fire you!")
            Exit Algorithm
        end
        numInUse = numInUse − 1
    end
end
print("Sufficient Rooms! Good Hotel! Rate you 5 stars 😘")
```

By sorting them respectively, we can generate a sequence of operation. Perform it operation by operation and trace the number of rooms in use on each day. If the number exceeds the largest capacity k-rooms, then the demands cannot be satisfied. Otherwise they will be satisfied.

The algorithm is described in Algorithm 1.

Ex. 2 **OUR ANSWER.**

(i) Firstly set x equals to e'_i , then $\prod_{i=1}^{n}(x-e'_i)$ should be equal to zero. If $P(x)$ equals to zero, then the value of $\prod_{i=1}^{n}(x-e_i)$ should be zero as well, which means there is one item in $[e_1,...,e_{ni}]$ whose value is equal to e'_i . Apply this method repeatedly for x from e'_1 to e'_{ni} , we can obtain the conclusion that if the polynomial holds for $[e'_1,...,e'_{ni}]$, then $[e_1,...,e_{ni}]$ is a permutation of it.

We can do the same things for $[e_1,...,e_{ni}]$, which provides the conclusion that $[e'_1,...,e'_{ni}]$ is the permutation of $[e_1,...,e_{ni}]$. To sum up, the condition that this polynomial holds, is the necessary sufficient condition.

(ii) $p-1$ should be larger or equal than e_{ni} and n, assume the largest value among $[e_1,...,e_{ni},e'_1,...,e'_{ni}]$ is e_{ni} . To ensure $P(x) \mod p \equiv 0$, the value of x, e_i, e'_i should be the same. since $[e_1,...,e_{ni}]$ is not the permutation of $[e'_1,...,e'_{ni}]$, the values that e_n equals to e'_n is less than the number of he items in $[e_1,...,e_{ni}]$. Besides, x is less than $p-1$. So, the possibility K that the result of evaluation is zero is less than $\frac{n}{p-1}$.

Meanwhile, $p-1$ should be also larger than $\frac{n}{\varepsilon}$, which shows $\varepsilon \geq \frac{n}{p-1}$. To sum up, the possibility K is at most ε

Ex. 3 **Our Answer.**

(i) Denoting the original stack S, and the two additional stacks S_1 and S_2. The process can be described as Algorithm 2.

Algorithm 2: Reverse stack

while *S is not empty* **do**
    S_1.`push(`S.`pop_top)`
end
while *S_1 is not empty* **do**
    S_2.`push(`S_1.`pop_top)`
end
while *S_2 is not empty* **do**
    S.`push(`S_2.`pop_top)`
end

(ii) Denoting the original stack S, and the queue Q. The process can be described as Algorithm 3.

Algorithm 3: Reverse stack again

while *S is not empty* **do**
    Q.`append(`S.`pop_top)`
end
while *Q is not empty* **do**
    S.`push(`Q.`pop_top)`
end

(iii) Assuming we have N elements $\{e_1, e_2, \cdots, e_N\}$ and two stacks S_A and S_B. The initial status of S_A is denoted as S_{A0}. The elements $e_i (1 \leq i \leq N)$ are placed in S_A initially.

The thing is, we try to copy S_A into S_B with the order of elements unchanged (making $S'_B = S_{A0}$); and then by popping each element in S'_B out and pushing back in S_A, we make the top element of $S'_B\prime$ the bottom one of the S'_A, i.e. the i-th in S'_B becomes the $(n-i+1)$-th in S'_A. Finally S'_A is the reversed from its original.

Define an operation MOVE, which one by one pops the top $(n-1)$ elements from S_A and pushes them into S_B, and store the n-th element (the bottom one) in the extra variable `temp`. The operation is described in Procedure MOVE.

Before MOVE($i-1$) is called, which indicates i times of calling MOVE, we will get the original stack with elements $[e_1, \cdots, e_{i-1}]$ (from top to bottom) and the extra stack with $[e_i, \cdots, e_n]$. Since a stack satisfies LIFO principle, and keep iterating and calling MOVE(n), MOVE($n-1$), ..., MOVE(1) we will finally get $S_B = S_{A0}$. Then pop all elements in S''_B and push them back to S''_A, we obtain S''_A is the reverse of S''_B, i.e. the reverse of S_{A0}. Algorithm 4 describes the whole operation.

Procedure MOVE(n)

```
/* Denoting $S_A$ as OriginalStack and $S_B$ ExtraStack. */
/* We are to copy the front $n-1$ elements into ExtraStack, reversely. */
/* Before MOVE($n$) is called, OriginalStack has $n$ elements. */
if $n = 1$ then
    ExtraStack.push(OriginalStack.pop_top)
    Exit call
end
while OriginalStack is not empty do
    ExtraStack.push(OriginalStack.pop_top)
end
/* Now we store the last element popped into ExtraVar. */
ExtraVar ← ExtraStack.pop_top
numElementsPopped ← 0
while numElementsPopped < $n$ do
    OriginalStack.push(ExtraStack.pop_top)
end
ExtraStack.push(ExtraVar)
Call MOVE($n-1$)
```

Algorithm 4: Reverse the Stack

```
/* Denoting $S_A$ as OriginalStack and $S_B$ ExtraStack. */
$n$ ← OriginalStack.numitem
Call MOVE($n$)
while ExtraStack is not empty do
    OriginalStack.push(ExtraStack.pop_top)
end
```