CSC263 Assignment 3

Akhil Gupta, 1000357071

November 2015

1. Give an algorithm for the following problem. The input is a sequence of n numbers $\{x_1, x_2, \ldots, x_n\}$, another sequence of n numbers $\{y_1, y_2, \ldots, y_n\}$, and a number z. Your algorithm should determine whether or not $z \in \{x_i + y_j \mid 1 \leq i, j \leq n\}$. You should use universal hashing families, and your algorithm should run in expected time $O(n)$.

   Answer: We need to check whether $z = x_i + y_j$. In order to accomplish this, we can use universal hashing. We can insert all $\{x_1, x_2, ..., x_n\}$ into a hash table T and check whether $z - y$ is in T. Hence, we need to use the insert and search functions i.e. Insert(T, x) and Search$(T, z - y)$. From lecture, we know that the expected time for a search is $O(1)$ and insert is also $O(1)$. Since we have n numbers in $\{x_1, x_2, \ldots, x_n\}$ and n numbers in $\{y_1, y_2, \ldots, y_n\}$, the expected time for insert is $n * O(1) = O(n)$ and search is $n * O(1) = O(n)$. In our algorithm, we need to insert all $\{x_1, x_2, \ldots, x_n\}$ into a hash table T and search through all $\{x_1, x_2, \ldots, x_n\}$ for $z - \{y_1, y_2, \ldots, y_n\}$ to check if it is in T. Therefore, the runtime is $O(n) + O(n) + O(n) = 3O(n)$ which is approximately $O(n)$

2. (a) What is the worst-case time complexity of a single operation in a sequence of m EnQueue and DeQueue operations? Derive matching upper and lower bounds. That is, define an initial situation by describing what H and T look like at the start, and then define a sequence of m operations, where the sequence consists of EnQueue's and DeQueue's. Then show that one of the operations in the sequence (probably the last operation) will have the claimed worst-case time. For the upper bound, show that no operation in any m-operation sequence can ever take more time than the claimed worst-case time.

   Answer: The worst-case time complexity of Enqueue in a sequence of m operations is $O(1)$ because all Enqueue has to do is push the element onto the stack T which takes constant time. The worst-case time complexity of Dequeue in a sequence of m operations is $O(n)$ where n is the number of elements on the stack. The worst case occurs when we need to pop all n elements from H and push them into another stack T and then pop H. This takes $2n + 1$ operations, therefore, runtime is $O(n)$.
   Initially H and T are empty stacks with each 6 slots. H = x x x x x x and T = x x x x x x
   Define a sequence of m operations: Enqueue(Q, 1), Enqueue(Q, 2), Enqueue(Q, 3), Enqueue(Q, 4), Enqueue(Q, 5), Enqueue(Q, 6), Enqueue(Q, 7), Enqueue(Q, 8)
   The above operations take 8*$O(1) = O(1)$ constant time
   After these operations: T = 6 5 4 3 2 1 and H = 7 8 x x x x
   Continuing the sequence of m operations: Dequeue(Q), Dequeue(Q), Dequeue(Q), Dequeue(Q), Dequeue(Q), Dequeue(Q), Dequeue(Q), Dequeue(Q)
   Now we encounter a problem, after the first 2 dequeues, stackEmpty(H) returns true which means H is empty. According to our code, if H is empty, we transfer the items of T to H by popping each item of T and then pushing it into H and then pop H.
   Note that stackEmpty(H) takes $O(1)$ time and the first 2 dequeues take 2*$O(1) = O(1)$ constant time
   After the first 2 dequeues: H = x x x x x x and T = 6 5 4 3 2 1
   After transferring the items of T into H, we have T = x x x x x x and H = 6 5 4 3 2 1.

The above operation took $O(1)$ time to call stackEmpty(H) + 6*$O(1)$ time to pop all the elements in T + 6*$O(1)$ time to push all the elements into H.
The remaining 6 dequeues take 6*$O(1)$ time. Therefore, in total we have 8*$O(1)$ + 2*$O(1)$ + 1*$O(1)$ + 6*$O(1)$ + 6*$O(1)$ = 23*$O(1)$ = $O(1)$.
We can see that the last operations (Dequeues) take the longest time. In the worst-case if H is empty, we might have to pop n elements from T = n*$O(1)$ then push those n elements into H = n*$O(1)$ then pop H = n*$O(1)$. Therefore, no operation in any m-operation sequence can take more than $O(n)$ worst-case time.

(b) Use the accounting method to prove that the amortised time complexity of each operation in a sequence of m EnQueue and DeQueue operations is $O(1)$.

To solve this problem, first give a credit scheme indicating how many credits to allocate to each EnQueue and DeQueue operataion. Secondly, state the credit invariant, and thirdly, prove the credit invariant.

Answer: Since we are implementing a queue with 2 stacks, notice that each element will be in 1 stack exactly once i.e. each element will be pushed twice and popped twice. We charge \$1 per push, \$1 per pop and \$1 to check whether a stack is empty or not. Hence, we need to allocate at least \$5 for the Enqueue operation. Since H and T are initially empty, we have to begin by pushing an element into the queue. We charge \$1 for this. Therefore, the element spent \$1 and stored \$4. We continue to add a series of elements into the queue. Each element will have \$4 credit. Now we perform the dequeue operation, first we need to check if H is empty or not, which costs \$1. Then we have to pop all the elements in T and push them into H and then pop H which costs \$3 in total (pop + push + pop). Hence, it costs \$4 in total to perform the dequeue operation. We have enough money since each element has \$4 credit on it from the enqueue operation. Therefore, the bank never goes broke.
Credit Invariant: Elements in H have \$3 stored and elements in T have \$4 stored.
Prove credit invariant:
Base case: since the stacks are initially empty, the invariant holds.
Inductive step: We have 2 cases: 1) when H is empty, 2) when H is not empty.
1) When H is empty: we have to pop all the elements in T and push them into H then pop H. Each element in T has \$4 stored on it (\$5 initially, but spent \$1 to push into T). Before pushing the elements into H, we check if H is empty. In our case, assume H is empty in inductive step. Therefore, each element spends \$1 to perform stackEmpty. Now they have \$3 stored. Now, all the elements are popped from T and pushed into H, therefore, each element spends \$1 to pop and \$1 to push. Now, they have \$1 stored. To perform the DeQueue operation, each element in H has to be popped. Hence, each element spends the remaining \$1 they have, therefore, we never go broke.
2) When H is not empty: while performing the DeQueue operation, we simply have to pop H which costs \$1. Suppose we perform many EnQueue operations until T is full, we continue pushing the elements into H. This does not cost any additional cost and the credit invariant is maintained.
In other words, for a sequence of m EnQueue and DeQueue operations, the amortized cost per operation is 4, which is $O(1)$.

3. Recall that the doubling method enables the implementation of a stack without placing a limit on the size of the stack, such that the amortized complexity of each operation is $O(1)$. Every time the array gets full, a new array is allocated whose size is twice the size of the old array, and the old array is copied to the new array.

(a) Suppose we change the implementation so that the size of the new array is 3/2 times the size of the old array. What is the time complexity of a sequence of m operations in the worst-case? Justify your answer.

Answer: We need to implement an array is 3/2 times the size of the old array. We begin with an empty array. Therefore, the first expansion occurs on an empty array. Now, the array size is 3/2. After a sequence of m operations, we need to again multiply the size of the array by 3/2 to make room for new elements. Now, the array size is 9/4 and so on. In general, the total time for a sequence of m operations is
$T(m) = m$ operations $+m + 3m/2 + 9m/4 + 27m/8 + ... + m(3/2)^{k-1}$
$= m + m(1 + 3/2 + 9/4 + 27/8 + ... + (3/2)^{k-1})$
Let $s =$ sum of $1 + 3/2 + 9/4 + ...$
Therefore, $T(m) = m + ms > m$
$T(m) = O(m)$
Therefore, the worst-case time complexity of a sequence of m operations is $O(m)$.

(b) Suppose we change the implementation so that the size of the new array is 50 plus the size of the old array. What is the time complexity of a sequence of m operations in the worst-case? Justify your answer.

Answer: We need to implement an array that expands by a constant $c = 50$. We begin with an empty array. Therefore, the first expansion occurs on an empty array. $0 + c = 50$. We expand the array by 50. Then after a sequence of m operations, specifically the 50th push operation on the stack, we need to expand the array by 50 again. Therefore, $50 + c = 100$. We can calculate the average cost per push, which is, $(50 + c)/50 = (50 + 50)/50 = 2$ operations per push. Then after another sequence of m operations, specifically the 100th push operation on the stack, we need to expand the array by 50 again. Therefore, $100 + c = 150$. We can calculate the average cost per push again, which is $(100+c+2c)/c = (100+50+100)/100 = 2.5$ operations per push. Similarly for 150 pushes, $(150 + c + 2c + 3c)/150 = (150 + 50 + 100 + 150)/150 = 3$ operations per push and so on. In general, the total time for a sequence of m operations is
$T(m) = m + c + 2c + 3c + ... + c * m/c$
$= m + c(1 + 2 + 3 + ... + m/c)$
$= m + c(m/c(m/c + 1))/2$
$= m + (m^2/c + m)/2$
$= O(m^2)$.
Therefore, the worst-came time complexity of a sequence of m operations is $O(m^2)$.