(1) **(Stack with Backup)** (20 points) Suppose you want to support a stack that has the operations Push, Pop, and Multipop as discussed in class, as well as the new operation

- Backup(S), which writes a copy of the entire contents of stack S to a file for archiving. (Backup does not alter S.)

Suppose that the size of the stack never exceeds k, and that Backup is called after every k operations on the stack.

Show that under these conditions, Push, Pop, Multipop, and Backup all take $O(1)$ amortized time, independent of k. Use the *accounting method* for your analysis.

1) We want to measure the total time taken for the number of pushes, pops, and backups. Count number of push, pops and multipops

2)

operation	Real time t_i	Amortized time a_i
Push	1	4
Pop	1	0
Multipop	$\min\{i, n\}$	0
back up	n	0

$\}$ $O(1)$

At any point the number of operation calls (k) would be at least the number of elements on the stack (n) because we can have at most k calls to push so the min of k and n will always be n and the runtime of backup will be the number of elements on the stack which is at most n. So the realtime of backup $= O(n)$

3) We can then give backup an amortized time of $O(0)$ because push is now accounting for the realtime cost of backup. Whenever we push we use 1 unit but get 4 back so we can use one to cover the cost of the operation then store 1 of the credits on the element just added then the other two on the stack itself or first element pushed to the stack. We can only pop (including the pops called in multipop) a number of times equal to the number of items pushed on the stack and never more so every call to pop (including the pops called on multipop) will be covered by 1 out of the two additional credits on every element in the stack.

4) Backup will be covered by the credits that are placed on the stack structure or bottom most element. We know these credits will always be greater than the number of elements in the stack because every push will add 2 credits to the stack. If all k operations between backups are pops then the cost of backup will be 0 so it doesn't matter how many credits are on the stack. If all k operations are pushes then we know the stack will hold twice as many credits as elements so the cost of backups will be covered. Due to there only being allowed k operations between backups and a max stack size of k elements we will either have an even number of pushes to pops, more pushes than pops, or more pops than pushes. If we have an even number of pushes to pops then there will be $k/2$ pushes, but each push adds 2 credits to the stack for backups so it will add $2k/2$ credits which is k (the max number of elements allowed in the stack) so all elements will be covered for a backup. If there are more pushes than pops then we will have

between $k/2 <$ pushes $\leq k$ so the number of credits will be $k <$ credits $\leq 2k$ which is always $\geq k$ so all elements will be covered for a backup. If there are more pops than pushes there will either be enough pushes to empty the stack so the credits needed for backup will be zero or we will push enough credits to cover the remaining elements left in the stack.

Ex. $k=50$ with 40 pops and 10 pushes. Lets say the stack was full so we do 40 pops and end up with 10 elements left. We will also do 10 pushes which will add 20 credits to the stack which will cover the cost of backing up 20 elements.

When there are more pops the pushes called then their will be a number of pushes called equal to the size of the stack minus the number of pops called at max so we will always have credit for backups when these new pushes are called.

The total amortized time is $O(4) + O(0) + O(0) + O(0) = O(5) = O(1)$

(2) **(Simulating a queue using stacks)** (30 points) Show how to implement the *queue* data structure by using two *stacks*, so that the amortized time for queue operations in the stack-based implementation matches their worst case time in a standard queue implementation. More specifically, show how to implement the operations

- `Put`(x, Q), which adds element x to the rear of queue Q, and
- `Get`(Q), which removes the element x on the front of queue Q and returns x,

so that both operations run in $O(1)$ amortized time. Use the *potential function method* for your analysis.

In order to implement a queue using two stacks we will:

- Put is called:
  - push to stack 1
- Get is called:
  - both stacks are empty → throw an error
  - stack 2 is empty:
    - pop each element off stack 1 and simultaneously push them onto stack 2 then pop the top element of stack 2
  - stack 2 isn't empty:
    - pop the top element of stack 2

Let $\Phi(D) := 2 \times$ height of stack 1

Then $\Phi(D_0) := 0$

$\Phi(D_i) \geq 0 + 2i$

measure real and amortized times:

$$a_{put} = t_{put} + \Phi(D_i) - \Phi(D_{i-1})$$

$$= 1 + n - (n-2) = 3 \rightarrow O(1)$$

1: real time cost of put on a queue

n: has a potential that is 2 more than $\Phi(D_{i-1})$ due to adding 1 element to stack 1

$(n-2)$: we added 1 element to stack 1 and $\Phi(D) = 2 \times$ height of stack 1 so $\Phi(D_{i-1})$ will have a potential 2 less than $\Phi(D_i)$

$$a_{get} = t_{get} + \Phi(D_i) - \Phi(D_{i-1})$$

stack 2 is not empty: $1 + 2n - 2n = 1 \rightarrow O(1)$

1: real time cost of a get on a queue

$2n$, $2n$: The height of stack 1 never changes

$$t_{get} + \Phi(D_i) - \Phi(D_{i-1})$$

stack 2 is empty: $2n+1 + 0 - 2n = 1 \rightarrow O(1)$

$2n$: The potential before the get was called was twice the height of stack 1

0: After get is called when stack 2 is empty

= potential before get was called
was twice the hight of stack 1

After get is called when stack 2 is empty there will no longer be any elements in stack 1 so the potential $= 2 \cdot 0 = 0$

real time cost: $2n$ because we have to pop every element off stack 1 and then push every element onto stack 2
+1 because of the last pop on stack 2 to oldest element in the queue

total amortized time $= O(3) + O(1) + O(1) = O(1)$

(4) **(Deleting the larger half)** (50 points) Design a data structure that supports the following two operations on a set S of integers:

- `Insert`(i, S), which inserts integer i into set S, where i is not currently in S, and
- `DeleteLargerHalf`(S), which deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Show how to implement this data structure so both operations take $O(1)$ amortized time. Use the *accounting method* for your analysis.

1) We will be counting the number of inserts and DeleteLargerHalfs in order to determine the amortized time.

2)

Operation	Real Time	Amortized Time
Insert	n worst case	8
DeleteLargerHalf	n	0

$\}$ O(1)

3) For this problem we will use an array that grows and shrinks dynamically as its size gets larger and smaller. When inserting we will simply be appending the new element to the end of the array. If the new element is unable to fit in the array due to it being at max capacity we will create a new array that is twice the size and copy the old elements over then append the new one. Inserts cost 1 to append to the end and give 8 so 1 will be used to cover the cost and 6 will go on the element itself and if any other element in the array has 5 credits on it we will put the remaining credit on that element. If all the elements have 6 or more we put the remaining credit on the newly added element. Whenever we need to double the array size, a new array of double the length is created and 1 credit is taken from each element in order to copy it over into the new array.
For DeleteLargerHalf we want to use the k^{th} smallest algorithm to find the element at position $\lceil |S|/2 \rceil$ and then we will partition all the other elements around that median. Both the k^{th} smallest algorithm and partitioning take $O(n)$ time so we will use 2 credits from each element to cover those costs. Next, we will delete all elements equal to and greater than the median. Deleting costs 1 so we will take one credit from the elements we are deleting. Then, any left over credits on any of the elements we are deleting will go over to an element that is not being deleted.

4) Since we assigned Inserts an amortized cost of 8, 1 credit will be used to cover the cost of an insert and 6 credits will go on the new element and the remaining credit will go on an element in the array with only 5 credits or the new element if there aren't any. Before any increases in the size of the array happen the max number of credits any element could have is 7 so in this case we know every element can cover the cost of copying itself over to the new, larger array. Since we are adding a credit to any element that has only 5 credits and the only way to get to 5 credits would be for an element to have 6 credits when an insert is called and the array needs to be resized we know whenever a resize is in order every element will have at least 6 credits because any element can only drop down to 5 immediatly after a resize. Then, we will only resize once the new array is full and since it is twice the size of the previous array we need to add just as many elements as there were when the previous resize occurred. Therefore, any element that has 5 credits will be guarenteed to be given another credit

by a new call to insert before the next resizing is required.
As for DeleteLargerHalf we know coming in to this call all elements have at least 5 credits from the analysis from above. When we call DLH all elements will pay to 2 credits to cover the k^{th} smallest algorithm to find the median and for partitioning around the median. So, all elements will now have 3 credits at a minimum. Next, when we delete all elements that are equal to or greater than the median we know all those elements have at least 3 credits so we take one from each and each is left with at least 2. Since we are deleting those elements we can give any remaining credits the have to the elements that aren't getting deleted. Since the smaller elements have at least 3 credits each on them and the ones being deleted have at least 2 and the number of elements being deleted is equal to the number of elements that are staying or 1 larger than the number of elements staying we are guarenteeing that all the elements that are staying are being given at least 2 credits therefore will have at least 5 credits when DLH is complete.

$|S| - \lceil |S|/2 \rceil = \lfloor |S|/2 \rfloor$ so the number of elements staying will always be less than or equal to the number being deleted.

Since each element had at least 5 credits before DLH was called and all elements in the array after DLH was called have at least 5 we know the net cost of a DLH is 0 on the remaining credits so we can call DLH any number of times back to back.

When the size of the array after a call to DLH is a quarter of the capacity of the array we can then contract the size of the array to half its original capacity.

Total amortized time = $O(8) + O(0) = O(1)$