Question 1:

The worst-case running time for bucket sort is $\Theta(n^2)$. We'll first understand how bucket sort works to find out the reason for this. Bucket sort creates an array of n linked lists (or buckets) and then segregates the input elements into those buckets accordingly. Following this, the algorithm uses Insertion sort to sort the elements in each linked list and finally concatenate all these linked lists to get the sorted version of the input array. Now here is where the problem of worst-case running time arises because in case the elements in any of the buckets are in descending order and, the number of elements in any of the buckets is very large (i.e) all the elements in the input array gets segregated to a single bucket, which makes the size of the single bucket as n, Insertion sort struggles here as the worst case running time of insertion sort is $\Theta(n^2)$ and an inversely sorted array is the main cause of the worst case running time of insertion sort.

Remedy:

To improve the worst-case running time of bucket sort to O(nlogn) while preserving its linear average-case running time, we can incorporate either merge sort or heap sort to sort the elements in each bucket. This is because both merge sort and heap sort have a worst-case running time of O(nlogn).

Question 2:

From the Question we can infer that X is equal to the number of heads in two flips of a fair coin.

Therefore, X can have the values 0, 1, or 2. This is because, during two flips of a fair coin, we can get either 2 heads, 1 head and 1 tail, or no heads.

$X = 0, 1, \ or \ 2$

Now the Probability formula $E[x]$ is,

$$E[x] = \sum x_i . P(x_i)$$

Here $P(x_i)$ denotes the probability of that event, (i.e.) the probability of getting that value of X.

So, the probability of getting 0 heads (or) $P(x_0) = P(Tails, Tails)$, (i.e.) getting two tails.

$P(x_0) = P(Tails, Tails) = P(Tails) * P(Tails)$
$(We \ know \ that \ the \ probability \ of \ getting \ Tails \ in \ a \ coin \ flip \ is \ 1/2.)$

$Therefore, \ P(x_0) = P(Tails) * P(Tails) = (1/2) * (1/2) = 1/4$

$$P(x_1) = P(Head, Tail) + P(Tail, Head)$$
$$= [P(Head) * P(Tail)] + [P(Tail) * P(Head)]$$
$$= [(1/2) * (1/2)] + [(1/2) * (1/2)]$$
$$= [1/4 + 1/4]$$
$$= 2/4$$
$$P(x_1) = 1/2$$

$$P(x_2) = P(Head, Head)$$
$$= P(Head) * P(Head)$$
$$= 1/2 * 1/2$$
$$P(x_2) = 1/4$$

Therefore,

$$E[x] = \sum x_i . P(x_i)$$
$$= [x_0 . P(x_0)] + [x_1 . P(x_1)] + [x_2 . P(x_2)]$$
$$= [0 * (1/4)] + [1 * (1/2)] + [2 * (1/4)]$$
$$= [0] + [1/2] + [1/2]$$
$$E[x] = 1$$

$$E[x^2] = \sum x_i^2 . P(x_i)$$
$$= [x_0^2 . P(x_0)] + [x_1^2 . P(x_1)] + [x_2^2 . P(x_2)]$$
$$= [0^2 * (1/4)] + [1^2 * (1/2)] + [2^2 * (1/4)]$$
$$= [0] + [1/2] + [1]$$
$$E[x^2] = 3/2$$

$$E^2[x] = E[x] * E[x]$$

Since we have calculated the value of $E[x]$ *already, we can just just substitute those values here.*

$$E^2[x] = 1 * 1$$
$$E^2[x] = 1$$

Therefore, $E[x^2] = 3/2$ *and* $E^2[x] = 1$

Question 4:

In order for the Randomized select function to make a recursive call to a 0-length array, there are two possible case for this:

1. Case 1: The lower bound and upper bound of the subarray should be equal. This means that p = q - 1. Then K will be 0 as we consider this to be a 0 element array. Therefore there will no elements to the left of the pivot in the subarray, which is the definition of k. But this recursive call can never happen because the if statement that encloses this recursive call states that i < k. This would mean that i is less than 0 (i.e.) i is a negative number. This can never be true because i is always positive because the rank of the smallest required element can never be a negative number.
2. Case 2: Again the lower bound and upper bound are equal. But in this recursive call, the equalities change, (i.e.)

$$q + 1 = r$$

$$k = q - p + 1$$

$$= r - 1 - p + 1$$

$$k = r - p$$

Therefore, for the second recursive call to be called, it should pass the if statement that states that i > k. This means that i > r - p. This is never true because we are trying to find the i^{th} element from a subarray that has less than i elements. This can never be true.

Question 5:

```
Randomized-select(A, p, r, i):
    while p < r:
        q = Randomized-Partition(A, p, r)
        k = q - p + 1
        if i == k:
            return arr[q]
        else if i < k:
            r = q - 1
        else:
            p = q + 1
            i = i - k
    return arr[p]
```

Explanation:

1. In order to make the recursive randomized-select to an iterative one, we first have to find the terminating condition of the recursion. The terminating condition will be the first if statement which checks if p is equal to r and if true, return the A[p].
2. We will use this terminating condition in order to break our iterative while loop. Therefore, we'll run the while loop until the p is less than r. When p is equal to r, we'll just return arr[p].
3. Inside each iteration of the while loop,
    3.1. First, we get q using the randomizedPartion function.
    3.2. Then we calculate k, the number of elements on the left side of the pivot in the sub array including the pivot.
    3.3. Then we check if i is equal to k, (i.e.) if the i^{th} smallest element that we need to find is equal to the k^{th} element of the subarray and if it is equal we just return the value A[k] as it is our needed smallest element.

3.4. If i is not equal to k, we first check whether i is less than k, that is the rank of our needed smallest element is less than the relative rank our pivot element. If i is less than k, we update the upper bound of our subarray to pivot - 1 because we know that our required element is located to the left side of the pivot element .

3.5. Else we check if i is greater than k, that is the rank our needed smallest element is greater than the relative rank of our pivot element. If i is greater than k, we update the lower bound of our subarray to q + 1 and i, the rank of our required smallest element, to be i = i - k because we know that our required element is located to the right side of the pivot element

3.6. After all this check, our while loop iterates again with the updated values of lower bound, upper bound and the rank our required smallest element.

Question 6:

```
StackEmpty(S):
    if S.top == 0
        return True
    return False

StackPush(S, x):
    if S.top == S.size:
        error "Stack Overflow"
    S.top = S.top + 1
    S[S.top] = x

StackPop(S):
    if StackEmpty(S):
        error "Stack Underflow"
    else:
        S.top = S.top - 1
        return S[S.top + 1]

EnQueue(S1, S2, x):
    StackPush(S1, x)

DeQueue(S1, S2, x):
    if StackEmpty(S2) == True:
        if StackEmpty(S1) == True:
            error "Queue Underflow"
        else
            While(StackEmpty(S1) == False):
                StackPush(S2, StackPop(S1))
    return StackPop(S2)
```

Explanation:

In order to implement a queue using two stacks, we can have two Stacks named S1 and S2 with standard stack Push and Pop Operations.

EnQueue method:

In the EnQueue method, we just get the element from the user and push it into our stack 1 (or) S1 using our standard stack push method which first checks whether the stack is full. If it is full, it returns a stack Overflow error. If it is not, we increment the top of the stack by 1 and add the new element to the new top of the stack.

DeQueue method:

In the DeQueue method, we first check if the Stack 2 (or) S2 is empty. If it is not empty, we just pop the element at the top of the stack 2 and return it to the user. If it is empty, we first check if the stack 1 is also empty and if it is, we return a Queue Underflow error as there are no elements in either of the stacks. If the stack 1 has elements in it, we use a loop, until the stack 1 is empty, during which we pop each element from Stack 1 and push it into Stack 2. This ensures that the elements in the Stack 2 are in reverse order of the elements in Stack 1 (i.e.) in reverse order of the input queue elements. This makes our dequeue operation accurate as this follows the FIFO (First In First Out) operation. After this while loop which reverses the order of elements, the first if which checks if Stack 2 is empty and if not pop the top element from Stack 2 makes sense as there may already be elements in reverse order in Stack 2 which when popped, follows the DeQueue rules of a Queue.

Analysis of Running Time:

The EnQueue Operation runs in $\Theta(1)$ time as the only operation done there is to push the element into Stack 1. There is no other processes done there.

The DeQueue Operation has a worst-case running time of $O(n)$ but in most cases runs in $\Theta(1)$. This is because it takes $O(n)$ running time only when Stack 2 is empty and requires the newly inserted elements from Stack 1. This triggers the loop to run which takes the elements from Stack 1 and inserts it into Stack 2 in the reverse order. But if the stack2 is populated, the DeQueue operation takes $\Theta(1)$ until Stack 2 is empty.

Question 7:

```
ReverseList(L):
    CurrentPrev = Null
    CurrentNext = Null
    node = L.head
    CurrentHead = L.head
    while node != Null:
        CurrentNext = node.next
        node.next = CurrentPrev
        CurrentPrev = node
        node = CurrentNext
    L.head = CurrentPrev
    L.tail = CurrentHead
```

Explanation:

1. First we initialise the values accordingly (i.e.) node is set to the current List's Head, CurrentNext and CurrentPrev is set to null. Also we have a variable CurrentHead to keep track of the original head of the Linked List.
2. Then we start to loop through the LinkedList with a terminating condition of the node should not be null.
3. During each iteration of the linked list, we do the following steps:
    3.1. Set the CurrentNext variable to the next node to the current node. This is done because after we change (or) reverse the next of the current node to the previous node, there will be no way for us to reach the next node as this is a singly linked list.
    3.2. Now we can change the current node's next to the CurrentPrev node. This is the step where we reverse the next of the current node to the previous node. In the first iteration, the

CurrentPrev is still initialised to Null, so the next of the first node will be null.

3.3. Now we set the CurrentPrev to the current node. This step is important because we need this to set the next of the CurrentNext node as we have no way of coming back to this node.

3.4. Finally we move the node variable(current node) to the CurrentNext node and proceed to the next iteration of the loop.

4. After we reverse all the links in the linked list, we now have to set change the head and tail of the linked list.

4.1. The head of the linked list can be set to the CurrentPrev node. This is because during the last iteration, the node variable will be null(after the last node), and the CurrentPrev will be pointing to the node before that, (i.e.) the last element of the linked list. Since we have reversed our list, the last node will now be the head of our reversed linked list.

4.2. The tail of the linked list can be set to the CurrentHead variable that we initialised at the beginning of our function which houses the original head of the unreversed linked list. Since we have reversed our list, the head node (or) the first node will now be the tail of our reversed linked list.

5. This whole process takes only $\Theta(n)$ running time because we only traverse through the linked list of n elements once and not more.