


**دانشکده‌ی علوم ریاضی**

**ساختمان داده** مدرس: دکتر شهرام خزائی

# تمرین سری سه

*نام و نام‌خانوادگی : امیر ملک حسینی* *شماره دانشجویی : ۴۰۱۱۰۰۵۲۸*

## پرسش ۱

we suppose that size of the list is n. Then we define two pointers i , j and set i to start from index 1 of list(suppose that start index is 1 in the list) and j to start from last index of list.
The last index of the list is size of the list so first we create a loop and iterate on list to find out size of the list.
In every state of while we check if j and i has passed each other, if not we compare elements i , j with each other and if they match, continue the while loop. if they don't it means that the list is not palindrome so we return false.
Because we have two loops with size n that are separate from each other, running time of algorithm is $O(n + n) = O(n)$.
This is the code for this algorithm:

```
function check(list):

   k = 0
   size = 0

   while(list[k].next != null)
      size++

   i = 1
   j = size

   while(i < j)

      if( list [i] = list[j] )
         i++, j--
      else
         return false
```

```
return true
```

## پرسش ۲

We know that each object in a singly linked list has a ”key” value that refers to object’s value itself , and a ”next” value that refers to next object of current object in the list. We know that we have a cycle in a list if and only if we can reach to an object of list after going through multiple states of the list.So if we want to check whether we have a cycle in list, we can assign first element in least as head of the list and then iterate on the list and search for ”next” value of some object be the head object.
If yes, then we have a cycle in the list.
The algorithm would be like this:

```
function check(list):

   head = list[0]

   for i in range list:

      if( list[i].next = head.key )
         return true

    return false
```

## پرسش ۳

## پرسش ۴

## پرسش ۵

The algorithm is like this:
First we create a min-heap with length k and at first we insert first elements of each list.Because lists are sorted if we apply Min-Heapify to the heap, the top element of the heap is the minimum of all lists.
So we pop it from the heap and insert it to final list. Then if we suppose that the heap knows which element came from which list( One way is that we give each list a specific index and heap store the index in a constant memory each time we push an element from a list), therefore we pop the top element and insert next element of

popped element's list to the heap.If the list has no more elements we do nothing and the heap will have $k-1$ elements and it will be fine.
In each use of Min-Heapify we find the smallest value between elements in the heap and insert it to final list, therefore by repeating this algorithm until the heap is empty we can sort all $n*k$ elements of all k lists in ascending order.
The time complexity of this algorithm is O(nk log k) because:
We insert and extract n elements from the heap k times and the children's subtrees each have size at most $2k/3$ and therefore we can describe the running time of Min-Heapify by the recurrence: $T(k) \leq T(2k/3) + \Theta(1)$
The solution to this recurrence, by case 2 of the master theorem is $T(k) = O(logk)$
So the overall time complexity of the algorithm is O(nk log k).
Here is the Min-Heapify function:

```
function Min_Heapify (arr, n, i):

l = 2 * i + 1
r = 2 * i + 2
smallest = i

if l < n and arr [l] < arr [smallest]:
    smallest = l

if r < n and arr [r] < arr [smallest]:
    smallest = r

if smallest != i:
    swap (arr, i, smallest)
    Min_Heapify (arr, n, smallest)
```

## پرسش ۶

آ)

The algorithm is this:
If we suppose that there is n elements in S1 , we introduce a variable "i" in a while loop and it goes from 1 to n.
In each iteration of outer loop:

1- Pop the top element of S1 and push it to a constant memory called "temp"
2- Create a new loop "1" from $n-i-1$ to 1 and pop these elements from S1 and push

them to S2
3- Push the temp element to S1
4- Create new loop "2" and pop elements in S2 and push them again to S1

In this algorithm the outer loop will be executed n times and each of inner loops will be executed n times as well. So the time complexity of the algorithm is O($n * 2n$) and that is equal to O($n^2$)
The algorithm procedure is like this:
The whole point is to reverse S1 elements using outer and inner loops and then in last iteration of outer loop( when i is equal to n ) pop reversed elements from S1 and push them to S2. So it will be pushed in the same order that it was placed in S1 at first.
Step 1 of algorithm pop the top element of S1 and push it to "temp" and after when step 2 is executed, the explicit elements of S1 is pushed to S2.In step 3 we push back the temp element to S1 and it is like that we reversed order of this element in S1 and step 4 brings back the elements from S2 to S1 in the same order that it was but with this difference that the top element of previous loop step is now reversed.
If we do these loop, because in each outer loop iteration the value of "i" is increasing so the elements that were reversed in previous iterations never be moved again in other iterations. It's like we choose $n-i-1$ elements( except the top elements that is already moved to "temp" ) and move them from S1 to S2 and from S2 to S1.
After reversing S1 elements is completed or in the other words "i" reaches the "n" in outer while loop, we break the loop so the loops will be eliminated and S2 has n elements with desired order.

**ب)**

This algorithm will not work if there is no extra memory.Because we only can pop and push to top of stacks . So without any extra memory we can't change stack's order and we need at most one bit extra memory. Therefore this is not possible.

## پرسش ۷

## پرسش ۸

For every stack we create a new stack called "$MaxElementStack$".It's like that the enhanced stack that I am suggesting is a stack object that has stack field in it called "$MaxElementStack$".
Pop and Push method of the enhanced stack is like a regular stack so it already has O(1) time complexity so we should check FindMax.
The procedure of enhanced stack is like this:
When we push a new element to stack, we compare the new element with the last element of "$MaxElementStack$" 's element. If the new one is bigger, we push the new item to "$MaxElementStack$" as well as adding it to the main stack, if not; we push the

last element of "$MaxElementStack$" to "$MaxElementStack$" again. And if we pop the last element of the main stack, we pop the last element of "$MaxElementStack$" as usual.
By doing this if we want the max element of the main stack, we just get that in O(1) by getting the last element of "$MaxElementStack$". We prove by induction that the last element of "$MaxElementStack$" is always the max element in main stack:
At first when the stack is empty the "$MaxElementStack$" of it is empty too. When we push a new element in the stack, we push it into "$MaxElementStack$" as well, because the stack was empty, so the new element that we pushed into stack is the max element so the enhanced stack works for base case of induction.
For induction step we suppose that the stack has n elements and nth element in "$MaxElementStack$" is the maximum. So when we insert a new element it could be the max for our stack or not.
Base on the procedure of enhanced stack, we compare the new element with last element of "$MaxElementStack$" that by induction step is already the maximum element in stack,and if the new one is bigger, it means that it is the new max element so we push it to "$MaxElementStack$" and the last element of "$MaxElementStack$" still is the maximum. If not; it means that the current max element still is the maximum element in stack and by push it again to "$MaxElementStack$" the last element of "$MaxElementStack$" is the max one. So by induction we conclude that the FindMax works in O(1) for pushing. Now we prove by induction that it works for popping as well:
If we have a stack with size two, when we pop an element from it(and "$MaxElementStack$" as usual), there will be a stack with size 1 and trivially it is the maximum. So it is true for base case.
For induction step we suppose that that the stack has n elements and nth element in "$MaxElementStack$" is the maximum. If we want to prove that then enhanced stack works precisely, the (n-1)th element of "$MaxElementStack$" should be the new maximum. and it is because:
When we pop an element from the stack, if it was the max one, by enhanced stack procedure we should pop the nth element of "$MaxElementStack$" as well and when we do that the (n-1)th element of "$MaxElementStack$" would be the last element in it and actually it is because it was the max before pushing the n th element, so when we removing the n th element it will be the max one again.
If we pop an element and it wasn't the max element we just pop a copy of true max element from "$MaxElementStack$" and the (n-1) th element in "$MaxElementStack$" still is the maximum.
‘ So we can conclude that the enhanced stack has pop,push,FindMax with O(1) time complexity
The push and pop is like normal stack and is:

```
Stack-Empty(stack):
```

```
   if stack.top == 0
      return true
    else return false


Push(stack,x):

   if !stack.top == stack.size:
      stack.top = stack.top + 1
      stack[stack.top] = x


Pop(stack):

   if !Stack-Empty(stack):

      stack.top = stack.top - 1
      return stack[stack.top + 1
```