**Student ID: 795622**

**Name: Dong Gao**

**UserName: dongg1**

**Answer 1**

**A**.

Heap just guarantees that each child's key is no greater than its parent (for max-heap), whose elements are not sorted actually. While BST makes sure that left subtree's elements are smaller than root, and right subtree's elements are larger than root, which could produce a sorted order by preforming the inorder traversal.

**B.**

No, we cannot. Heap is not a sorted structure, because it just guarantees that each parent node’s key is no smaller than its children’s, but we cannot make sure whether left or right child’s key is bigger. If we want to traverse a heap, we must sort its elements before, which, actually called heapsort, spends  time.

**Answer 2**

function TRIANGULABLE(Points S[0,1,…,n-1])

//the input is the coordinates of n points

//the output is true or false

if n < 3 then

return false

for i🡨0 to n-2 do

infiniteslope 🡨 0

k 🡨 0

slope[n] 🡨 null

for j🡨i+1 to n-1 do

if S[i].x=S[j].xthen

infiniteslope 🡨 infiniteslope+1

else

slope[k] 🡨 (S[i].y - S[j].y) / ( S[i].x - S[j].x)

k 🡨 k+1

if infiniteslope>1 then

return false

*//use the heapsort algorithm defined in Lecture 13, page 19*

Heapsort(slope[0,1,…,k-1])

for j🡨0 to k-2 do

if slope[j] = slope[j+1]

return false

return true



**Answer 3**

The basic idea is that extract the head of each linked list, and store them in an array. Execute the build heap algorithm (defined in Lecture 13, page 15) on this array, find out the minimum value, store this element in a new list, and use its next element to replace its position in the array. Then adjust the new array to a min-heap again (one sift down operation is ok), and repeat the above mentioned operation until all elements are stored in that new list. If the minimum value has no next point, then replace it with the last element in the array, and make array length minus one. Finally, we can get a single sorted list.

function MergeLists(List p1,List p2,…,List pk)

//the input is k sorted lists

//the output is a single sorted list with n elements

create List merge //build a new list to store the merge results

create array[k]

p 🡨 merge.head

for i 🡨 1 to k

array[i] 🡨 p[i].head

*//use the create heap algorithm* *defined in Lecture 13, page 15*

CreateMinHeap (array[1,…,k])

for k > 0 do //when the array(actually, the adjusted heap) is not null

p 🡨 array[1] *//array[1] is the minimum value of the heap*

p 🡨 p.next

if array[1].next != null

array[1] 🡨 array[1].next

else

array[1] 🡨 array[k]

k 🡨 k-1

*// just sift down root(array[1])of the heap, which has a log k complexity*

*//this operation defined in Lecture 13, page 14*

HeapSiftDown(array[1,…,k])

return merge

The running time, in the worst case, has the operations as follows:

1. Build an array to store all k lists’ head elements, cost time of .
2. Turn the array into a heap, bottom up, cost time of .
3. To get a sorted list of n elements, a heap of size k has to sift down its root for n-k times. So the time cost is .

So, the total run time is:

