Assignment 1

Siwei Tang (siweitang)
sw.tang@mail.utoronto.ca

Changyan Xu (xuchangy)
changyan.xu@mail.utoronto.ca

February 11, 2021

Reference

- https://www.geeksforgeeks.org/merge-k-sorted-linked-lists-set-2-using-min-heap/

Question 1

For the algorithm `hasTwentyOne(L)`, we assume there is at most one 21 in the list and that is equally likely to be in any of the n positions or not in the list at all. It means that

$$\begin{aligned} P(\text{21 appears at position 1}) &= P(\text{21 appears at position 2}) \\ &= ... \\ &= P(\text{21 appears at position k, where } 1 \le k \le n) \\ &= ... \\ &= P(\text{21 appears at position } n) \\ &= P(\text{21 does not appear in the list}) \\ &= \frac{1}{n+1} \end{aligned}$$

Since line 1, 3 and 4 only takes constant time, we count the two comparisons in line 2 as a parameter for time measurement.
If the only one 21 is at position 1, the probability of exiting at position 1 is $\frac{1}{n+1}$. It has 2 comparisons.
If the only one 21 is at position 2, the probability of exiting at position 2 is $\frac{1}{n+1}$. It has 4 comparisons.
...
If the only one 21 is at position k, the probability of exiting at position k is $\frac{1}{n+1}$. It has $2k$ comparisons. (where $1 \le k \le n$).
...
If the only one 21 is at position n, the probability of exiting at position n is $\frac{1}{n+1}$. It has $2n$ comparisons.
If there is no 21 in the list, the probability is $\frac{1}{n+1}$. It takes $2n+1$ comparisons, where 2 comparisons at each position for n iterations and 1 extra on "`j is None`" before exiting the while loop.

Average-case Runtime for `hasTwentyOne(L)`:

$$\begin{aligned} E[t] &= \sum_{\text{all cases}} t \times Pr[T = t] \\ &= Pr(\text{21 not in list}) \times (2n+1) + \sum_{k=1}^{n} Pr(\text{first 21 at position k}) \times 2k \\ &= \frac{1}{n+1} \times (2n+1) + \sum_{k=1}^{n} \frac{1}{n+1} \times 2k \\ &= \frac{2n+1}{n+1} + \frac{2}{n+1} \sum_{k=1}^{n} k \\ &= \frac{2n+1}{n+1} + \frac{2}{n+1} \times \frac{n \times (n+1)}{2} \\ &= \frac{2n+1}{n+1} + n \\ &= \frac{n^2+3n+1}{n+1} \end{aligned}$$

Question 2

Best Case Analysis:
The submission is illegal at position 1 in submissions. The `for` loop exits at line 5 for the first loop iteration. Since each iterations takes constant times, T(best case) $\in \Omega(1)$.

Worst Case Analysis: Split into two scenarios.
Scenario 1: illegal submission at position n
It exits at line 5 for the n^{th} loop iteration and runs n iterations. Since each iteration takes constant time, the total count is n.
Scenario 2: all submissions are legal
It exits at line 6 for the n^{th} loop iteration and runs n iterations. Since each iteration takes constant time, the total count is n.
$\Rightarrow$ T(Worse case) $\in O(n)$.

Average Case Analysis: two scenarios
Scenario 1: students are no more likely to pick someone from their own section than to pick a random student from the course.
700 * 0.2 = 140 students would like to work without a partner and 700-140 = 560 students would like to form groups, so totally there would be 560/2 + 140 = 420 groups.
$\Rightarrow$ The length of submission array is 420.
Since the solo worker group must be legal, so the illegal group can occur in the two-partner group which the amount is 280.
Denote S1 as the number of students who want to form group in section 1
Denote S2 as the number of students who want to form group in section 2
Denote S3 as the number of students who want to form group in section 3

$$S1 = 560 \times \frac{240}{700} = 192$$
$$S2 = 560 \times \frac{240}{700} = 192$$
$$S3 = 560 \times \frac{220}{700} = 176$$

Since the probability is independent of the position in the list and independent of the other values in the list, the probability (p) that a group contains an across-section partnership at each point in the submission list is:

$$\begin{aligned} p &= \frac{140}{420} \times 0 + \frac{280}{420} \times \left(\frac{192}{560} \times \frac{192+176}{560} + \frac{192}{560} \times \frac{192+176}{560} + \frac{176}{560} \times \frac{192+192}{560}\right) \\ &= \frac{1}{3} \times 0 + \frac{2}{3} \times \frac{816}{1225} \\ &= \frac{554}{1225} \approx 0.444 \end{aligned}$$

For $1 \le k \le n$, when the first illegal group occurs at position k, it take k iterations to exit the for loop. The probability to exit the algorithm at position k of the loop is $p(1-p)^{k-1}$. When there is no illegal partnership in the submission list, it take n iterations to exit the for loop. The probability to exit the algorithm for such case is $(1-p)^n$.

$$\begin{aligned}
E[t] &= \sum_{\text{all cases}} t \times Pr[T=t] \\
&= Pr(\text{no illegal group in submission list}) \times n + Pr(\text{first illegal group at position 1}) \times 1 + \\
&\dots + Pr(\text{first illegal group at position } n) \times n \\
&= (1-p)^n \times n + \sum_{k=1}^{n} (1-p)^{k-1} \times p \times k \\
&= (1-p)^n \times n + \frac{p}{1-p} \sum_{k=1}^{n} (1-p)^k \times k \\
&= (1-p)^n \times n + \frac{p}{1-p} \times \frac{(1-p)^{n+1}(n(1-p-1)-1)+1-p}{p^2} \\
&= (1-p)^n \times n + \frac{p}{1-p} \times \frac{(1-p)^{n+1}(-np-1)+1-p}{p^2} \\
&= (1-p)^n \times n + \frac{(1-p)^n(-np-1)+1}{p} \\
&= (1-p)^n \times (n-n-\frac{1}{p}) + \frac{1}{p} \\
&= \frac{1}{p} - \frac{1}{p} \times (1-p)^n
\end{aligned}$$

Plugging in $p = 0.444, n = 420$, we get $E[t] \approx 2.25$

Scenario 2: students are equal likely to choose partner from their own section and from other two sections.
$\Rightarrow$ P(choose in own section) = P(choose in other two sections) = $\frac{1}{2}$

$$\begin{aligned}
p &= \frac{140}{420} \times 0 + \frac{280}{420} \times \frac{1}{2} \\
&= \frac{1}{3} \\
\Rightarrow E[t] &= \frac{1}{p} - \frac{1}{p} \times (1-p)^n \\
&= 3 - 3 \times (\frac{2}{3})^{420} \\
&\approx 3
\end{aligned}$$

Question 3

(a) General approach with $\mathcal{O}(n)$ of merging the two arrays (i.e. A and B) into one sorted array X:

1. Iterate both arrays (i.e. A and B), and find the length of each (i.e. a and b)
2. By comparing the first index of A and B (i.e. index 0; the largest values of each array respectively), record the larger element in the result array. Let's say A's largest was just recorded. Move to the next largest element of the array A. A new comparison is then carried out between this new one in A and the old one in B. Repeat the process by iterating over both A and B until at least one array runs out.
3. Copy the rest of elements in the remaining array directly into the end of the result array. The result array is the sorted array X that we want.

(b)

Algorithm 1 Merge Two Sorted Arrays

Input: Two sorted arrays A and B in descending order
containing a and b elements respectively
Output: An sorted array X containing $n = a + b$ unique elements

```
1  $a, b \leftarrow 0, 0$
2  for $i = A[0], \ldots, A[-1]$ do
3      $a \leftarrow a + 1$
4  for $j = B[0], \ldots, B[-1]$ do
5      $b \leftarrow b + 1$
6  $result \leftarrow []$
7  $i, j \leftarrow 0, 0$
8  while $i < a$ and $j < b$ do
9      if $A[i] \geq B[j]$ then
10         $result \leftarrow result + [A[i]]$
11         $i \leftarrow i + 1$
12     else
13         $result \leftarrow result + [B[j]]$
14         $j \leftarrow j + 1$
15 if $i \geq a$ then
16     $result \leftarrow result + B[j :]$
17 if $j \geq b$ then
18     $result \leftarrow result + A[i :]$
19 return $result$
```

(c) Worst-case Analysis:

- Worst-case scenario: $a \geq b$
- we count comparisons as run-time measurement
- line 1, 3, 5 are assignments which take constant time.
- line 2 and line 4 are loops, which take a total $a + b$ iterations.
- again, line 6 and 7 are assignments taking constant time.
- for the while loop from line 8 to 14, it has at most $3b + 2$ comparisons
  * line 10, 11, 13, 14 are assignments, taking constant time.

    * for each iteration in the while loop, it takes a total 3 comparisons
    * before the loop exits, while loop evaluates 2 comparisons (i.e. $i < a$ and $j < b$), whereas $i < a$ passed but $j < b$ does not pass.
  - from line 15 to 19, it takes 2 comparisons in total.
  - Thereby, the algorithm takes $a + b + 3a + 2 + 2 = 4a + b + 4 = \boxed{n + 3a + 4}$ comparisons, which is of $O(n)$.
  - Or a number of element comparisons

(d) Best-case Analysis:
Assuming that a and b are large (but unknown) values.

  - **Case 1:** If the last element of A is greater than or equal to the first element of B, so during the while loop iterations, only line 9 - 11 would execute, and exits the while loop by violating $i < a$. The while loop iteration times depends only on length of A which is a, so running time is $\Omega(a)$.
  - **Case 2:** If the last element of B is greater or equal to the first element of A, so during the while loop iterations, only line 12 - 14 would execute, and exits the while loop by violating $j < b$. The while loop iteration times depends only on length of A which is a, so running time is $\Omega(b)$.

Best-case Running time: $\Omega(a)$ or $\Omega(b)$
Family of input cases:

  - For the best-case runtime is $\Omega(a)$, input family is the array A and B, where the last element of A is greater or equal to the first element of B.
  - For the best-case runtime is $\Omega(b)$, input family is the array A and B, where the last element of B is greater or equal to the first element of A.

(e) **We only count the element comparisons in all cases. (That's mean we only count comparison at line 9)**
The number of combinations is $\sum_{a=0}^{n} \binom{n}{a} = 2^n$
Since we can divide problem into two cases: A cleans first or B cleans first, since the condition that A has one element and B has one element is in fact the same analysis so we only need to do analysis on $\frac{2^n}{2} = 2^{n-1}$ combinations.

Consider X to be the merged array with length n, and let's say that the elements are integers from n to 1 in descending order.
If A is empty, it takes a total 0 comparison.
If A contains only integer n, it takes a total 1 comparison.
If A has two cases, contains integer $n - 1$, contains integer n and $n - 1$, both would take 2 comparisons. If A has 4 cases, contains $n - 2$, contains n and $n - 2$, contains $n - 1$ and $n - 2$, contains $n, n - 1$ and $n - 2$, it takes a total 3 comparisons.
And so on.

$$\begin{aligned}
E[t] &= \sum_{\text{all cases}} t \times Pr[T = t] \\
&= \frac{1}{2^{n-1}} \times 0 + \frac{1}{2^{n-1}} \times 1 + \frac{2^1}{2^{n-1}} \times 2 + \frac{2^2}{2^{n-1}} \times 3 + ... + \frac{2^{n-2}}{2^{n-1}} \times (n-1) \\
&= \sum_{i=1}^{n-1} \frac{1}{2^i} \times (n-i) \\
&= 2 \times (\sum_{i=1}^{n-1} \frac{1}{2^i} \times (n-i) - \frac{1}{2} \times \sum_{i=1}^{n-1} \frac{1}{2^i} \times (n-i)) \\
&= 2 \times (\frac{1}{2} \times (n-1) + \frac{1}{4} \times (n-2) + ... + \frac{1}{2^{n-1}} - (\frac{1}{4} \times (n-1) + \frac{1}{8} \times (n-2) + ... + \frac{1}{2^{n-1}} \times 2 + \frac{1}{2^n})) \\
&= 2 \times (\frac{1}{2} \times (n-1) - (\frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^{n-1}} + \frac{1}{2^n})) \\
&= 2 \times (\frac{1}{2} \times (n-1) - \sum_{i=2}^{n} \frac{1}{2^i}) \\
&= 2 \times (\frac{1}{2} \times (n-1) - (\sum_{i=0}^{n} \frac{1}{2^i} - 1 - \frac{1}{2})) \\
&= 2 \times (\frac{1}{2} \times (n-1) - (\frac{\frac{1}{2^{n+1}} - 1}{-\frac{1}{2}} - 1 - \frac{1}{2})) \\
&= 2 \times (\frac{1}{2} \times (n-1) - (\frac{1}{2} - \frac{1}{2^n})) \\
&= n - 2 + \frac{1}{2^{n-1}}
\end{aligned}$$

(f) Devise an algorithm with worst-case run-time $\mathcal{O}(n \log k)$

See the pseudocode Algorithm 2 in the next page.

Data Structure: k sorted linked list, and a max heap which can hold at most k elements

Part 1: Reform k sorted arrays into k sorted linked list

Part 2: Build a heap with the max value of each link list from the k linked lists. Insert (at most) k values one by one into the max heap.

Part 3: Extract the max from the max heap, and put it into the result array X. Add the next value of the popped node in the link list to the max heap with, if there exists one. Repeat the procedure for *ntimes* (until the max Heap is empty)

The result array X is what we want.

Algorithm 2 Merge k Sorted Arrays

```
Input: k sorted arrays in descending order containing n elements in total
Output: An sorted array X containing n unique elements
20 # Part 1:
   # transform k arrays into k linked lists
   # O(n) —- put n elements into nodes as part of linked lists
21 linkys ← []
22 for arr in k_arrays do
23     if arr then
24         link ← LinkedList()
25         link.first ← _Node(arr[0])
26         if arr[1:] then
27             curr ← link.first
28             for i in arr[1 :] do
29                 curr.next ← _Node(i)
30                 curr ← curr.next
31     linkys ← linkys + [link]
32 # Part 2:
   # Build a heap with the max value of each of the k linked lists.
   # Insert k values one by one into the heap
   # O(klogk) —- O(k) calls to MaxHeap.insert(value) and each one takes O(logk)
33 maxHeap ← MaxHeap()
34 for link in linkys do
35     maxHeap.insert(link.first)
36 # Part 3:
   # ExtractMax from the heap into the result array X
   # O(nlogk) —- O(n) calls to MaxHeap.extractMax() and each one takes O(logk)
37 X ← []
38 while maxHeap is not empty # for _ in range(n) do
39     popout ← maxHeap.extractMax()
40     X ← X + [popout]
41     if popout.next is not NIL then
42         maxHeap.insert(popout.next)
43 return X
```

Runtime Justification:

Part 1: (line 21-31) We put n elements into nodes as part of linked lists. It takes $\mathcal{O}(n)$ running time.

Part 2: (line 33-35) It takes $\mathcal{O}(k)$ calls to `MaxHeap.insert(value)` and each one takes $\mathcal{O}(\log k)$. This part takes $\mathcal{O}(k \log k)$ running time.

Part 3: (line 37-43) It takes $\mathcal{O}(n)$ calls to `MaxHeap.extractMax()` and each one takes $\mathcal{O}(\log k)$. This part takes $\mathcal{O}(n \log k)$ running time.

Summary: Putting the running time all together: $\mathcal{O}(n) + \mathcal{O}(k \log k) + \mathcal{O}(n \log k) \in \mathcal{O}(n \log k)$

Correctness Justification:

Part 1: (line 21-31) We make k arrays into k linked lists, which are stored in an array called `linkys`.

Part 2: (line 33-35) A max heap is created with at most k items, where each of the at most k items is from the head of the linked list.

Part 3: (line 37-43) It pull an item of the maximum each time from the heap into the result array X, following with a possible insertion of the next node item of the popped-out item from the linked list.

Summary: From part 2 and part 3, the total number of insertion into the heap from the linked lists is n, and the total number of `ExtractMax` from the heap to the result array X is also n. Thereby, X is of a total number of n elements, and the items are in descending order because of the `ExtractMax` operation.

Question 4

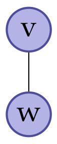
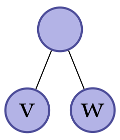
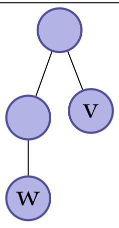
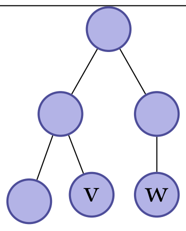
a) The pseudo-code for the algorithm is as followed:

Algorithm 3 Insert Last Node

```
Input: The reference to the current last node $v$, the insertion node $w$
curr ← v.parent
prev ← v
# Case 1: v is the root
if curr is NIL then
    prev.left ← w
    return
# Case 2: v is a left leaf
if curr.left is prev then
    curr.right ← w
    return
# Case 3,4: v is a right leaf
while curr is not NIL do
    # curr should start to move downwards in the right heap. (2nd part of Case 4)
    if curr.right is not NIL and prev.parent.right is not prev then
        prev ← curr
        curr ← curr.right
        break
    # curr should move upwards in the heap from v.parent to at most the root of the heap. (1st
      part of Case 3&4)
    else if curr.parent is not NIL then
        prev ← curr
        curr ← curr.parent
    # curr should start to move downwards in the left heap. (2nd part of Case 3)
    else
        break
# curr would never be NIL by now, because of the If-elseif-else in the above while loop.
# curr moves downwards in the following sub-heap. Assign the w to leftmost-leaf in the cur-
  rent sub-heap.
while curr.left is not NIL do
    curr ← curr.left
if curr.left is NIL then
    curr.left ← w
else
    curr.right ← w
return
```

Note:
The heap could never an empty binary tree, because a node v is given and exists. Additionally, due to the binary tree interface, an empty tree refers to a `null` object, which does not fit the interface with only methods `T.left`, `T.right` and `T.parent`.

The given algorithm had met the following corner cases:

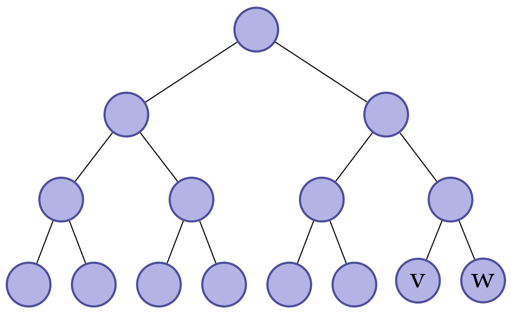
Case 1	Case 2	Case 3	Case 4
			
v is the root	v is an arbitrary left leaf	v is an right leaf of a complete tree where the last level is fully filled	v is an right leaf of a complete tree where the last level is not yet fully filled

b) Best case runtime analysis:
Assume the heap contains k items.
The input family of the best case is that v is an arbitrary left leaf in the heap. So all the algorithm needs to do is that make the $v.parent.right$ the w node, by first gain the information of v is a left leaf. So it only implements from line 51 to 53, costing a constant time because of the only involvement of a simple pointer comparison and an assignment (plus a return).

Hence, the best case runtime is $\Omega(1)$.

For other inputs, v and w do not share the same parent. So in order to insert w as the last node, the algorithm must traverse at least two nodes, including the parent of v and the parent of w. However, for the best case, since v and w share the same parent, so the algorithm only traverse one node which is the parent of v. So no other input can possibly do better.

e.g.



c) Worst case runtime analysis:
Assume the heap contains k items.
The input family of the worst case is that k items form a full binary tree as scenario 1 before insertion, where v is the rightmost leaf. In such a case, we can only insert w as the left child of the leftmost leaf which is before the insertion (i.e. node s). As we know, the height of the tree is $\lfloor \log_2 k \rfloor$. The insertion would traverse from the rightmost leaf v to the root and then from root to s, so it would traverse $2\lfloor \log_2 k \rfloor + 1$ or $2\times$ height $+1$. (e.g. height = 3, transverse nodes = $2\times$ height $+1 = 2\lfloor \log_2 15 \rfloor + 1 = 7$)
Taking Scenario 1 (ii) as an example,

- $k = 1$
- height = 0
- actual number of transverse nodes = 1

- most number of transverse nodes = $2\times$ height $+1 = 2\lfloor \log_2 1 \rfloor + 1 = 1$ = actual number of transverse nodes

Hence, the worst case runtime is $\mathcal{O}(\log k)$.

For other inputs, k items can only form a complete binary tree but not full (such as scenario 2). It would traverse at most $2\lfloor \log_2 k \rfloor$ or $2\times$ height.
Taking Scenario 2 (i) as an example,

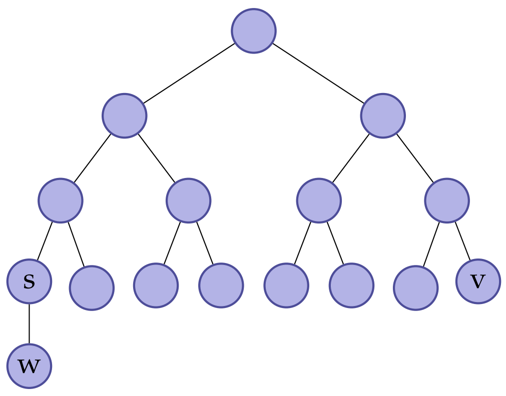
- $k = 11$
- height = 3
- actual number of transverse nodes = 6
- most number of transverse nodes = $2\times$ height $= 2\lfloor \log_2 11 \rfloor = 6 \geq 6$ = actual number of transverse nodes
- Although it still takes a runtime of $\mathcal{O}(\log k)$, it is not the worst in the worst case.

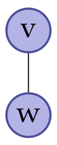
Taking Scenario 2 (ii) as an example,

- $k = 13$
- height = 3
- actual number of transverse nodes = 4
- most number of transverse nodes = $2\times$ height $= 2\lfloor \log_2 13 \rfloor = 6 \geq 4$ = actual number of transverse nodes
- Although it still takes a runtime of $\mathcal{O}(\log k)$, it is not the worst in the worst case.

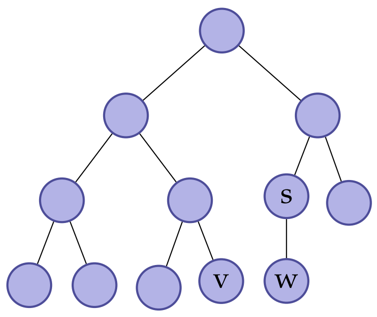
Thereby, no other input can possibly do worse.
e.g.

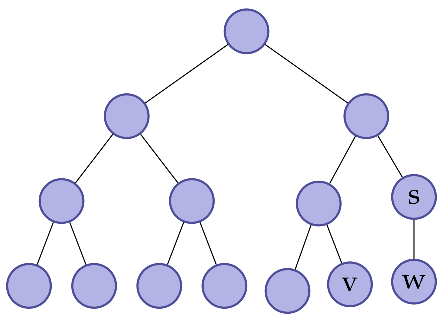
Scenario 1:

(i)


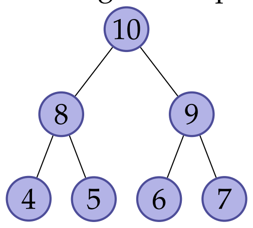
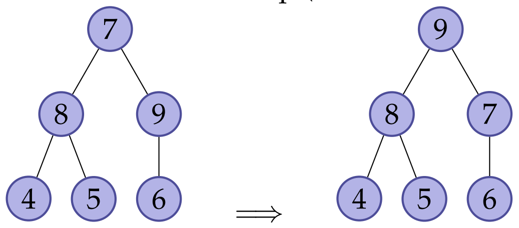
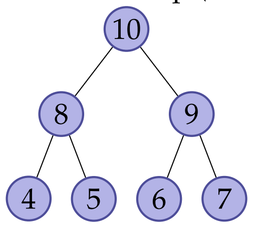
(ii)


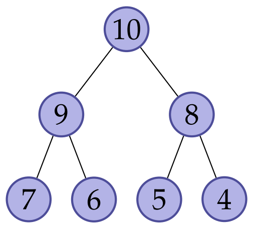
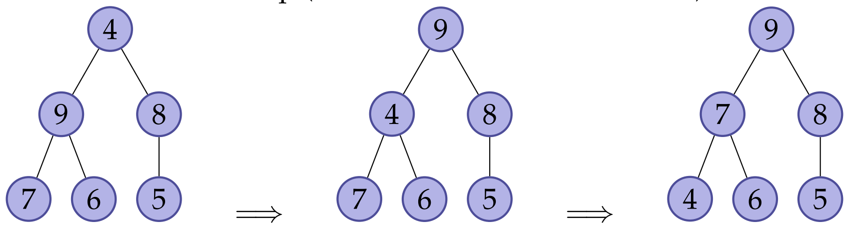
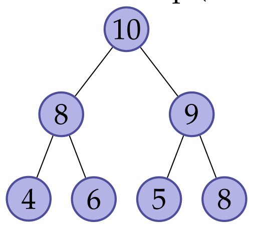
Scenario 2:

(i)


(ii)


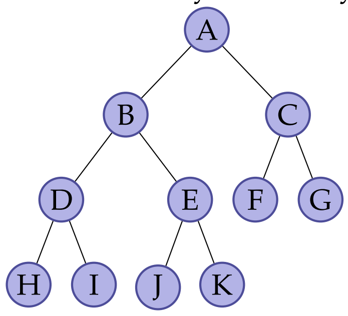
Question 5

1. a DIS as an array: [10, 8, 9, 4, 5, 6, 7]
   The original heap:
   
   The intermediate heap (after the call to `ExtractMax()`):
   
   The final heap (after the call to `INSERT(10)`):
   

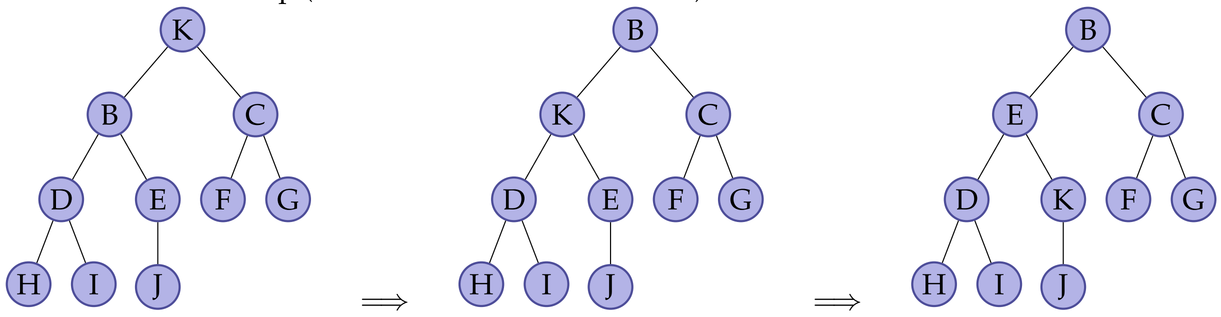
2. a non-DIS as an array: [10, 9, 8, 7, 6, 5, 4]
   
   The intermediate heap (after the call to `ExtractMax()`):
   
   The final heap (after the call to `INSERT(10)`):
   

3. Rule:
   For an arbitrary DIS binary max heap, all nodes in the route from the root (i.e. A) to the the rightmost node of the lowest row (i.e. K), are greater than the counterpart node of the same parent (if there exists) respectively.

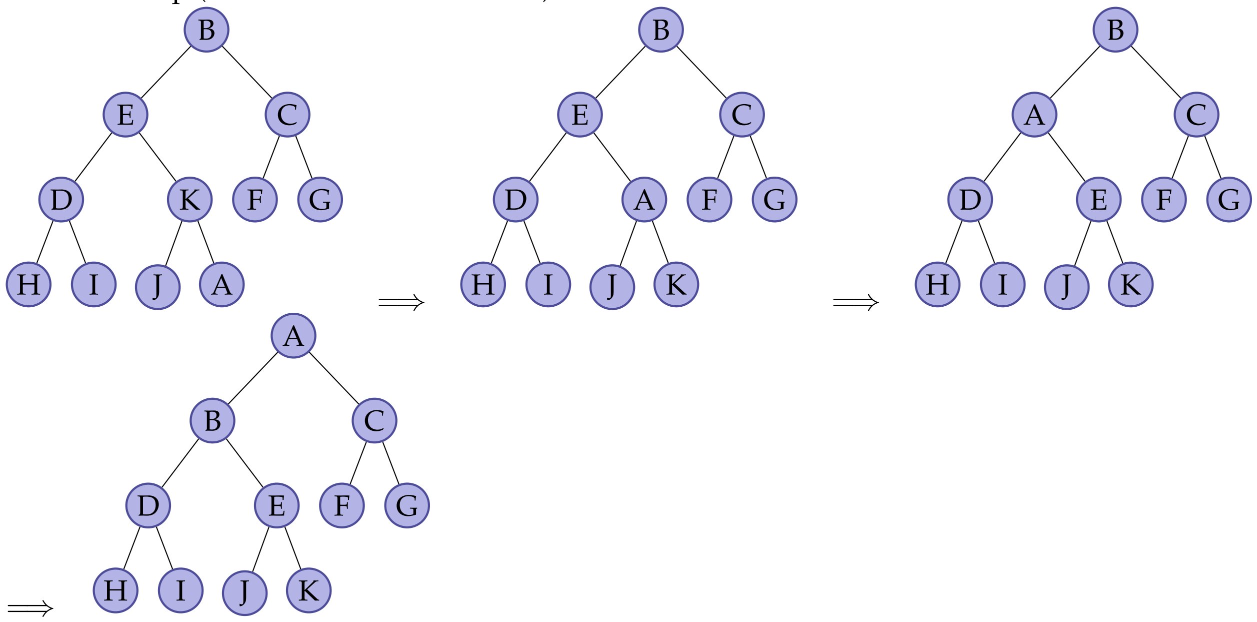
4. For an arbitrary DIS binary max heap,



The intermediate heap (after the call to `ExtractMax()`):



The final heap (after the call to `INSERT(A)`):

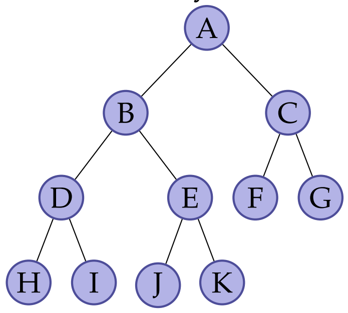


Justification:
A (binary) max heap is a nearly complete binary tree and the each node value is greater or equal to its immediate children. After calling to `ExtractMax()`, the root value is removed, and the rightmost node of the lowest row is filled into the root at first, then followed by the bubble down (or max-heapify) process. During the bubble down, we need to exchange the root with its immediate child if its immediate child is greater than the value of root. The just-mentioned procedure should be propagated down the tree until having made to a valid heap.

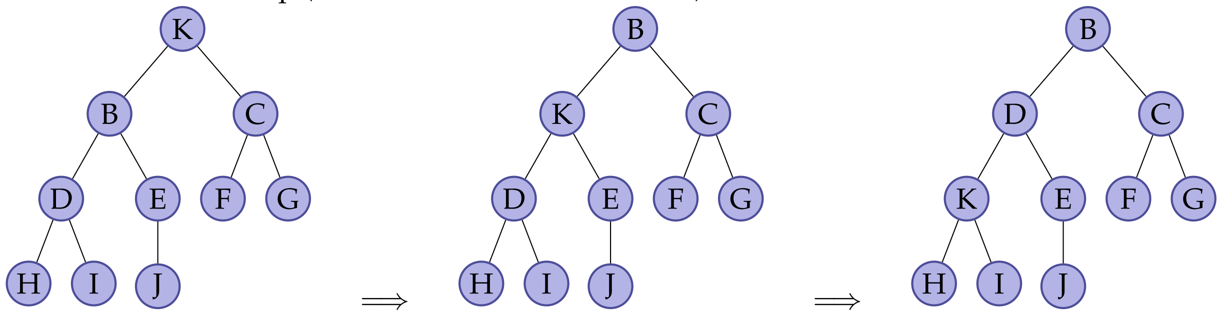
Meanwhile, if the heap satisfies the above-mentioned DIS heap rule, the route would be $ABEK$, where $A \geq B \geq E \geq K$ and $B > C, E > D, K > J$. As the newly promoted root K is smaller or equal to B and $B > C$, K may exchange value with B. Then it followed by value of position B may exchange value with position E. And the bubble-down stops because $J < K$ and K value now may just at position E.
After the call to `INSERT(A)`, A is first filled at the rightmost node of the lowest row, and then bubbled-up along the same route (i.e.KEB). At the end A is again the root of the max-heap tree, and the final heap is identical to the original heap. All in all, it is a DIS heap because the bubble-down route for `ExtractMax` is the same route as the bubble-up route for `Insert(A)`.

However, if a heap is not DIS which does not satisfy the above-mentioned rule, after the call to `ExtractMax`, the newly promoted root K may exchange values with C and on and on to form a distinct path of change (e.g. $(A)BD$ or others, but not $(A)BEK$ of the original heap). Whereas the bubble-up route for `Insert(A)` is still the $(A)BEK$ of the original heap. All in all, it is a non-DIS heap because the bubble-down route for `ExtractMax` is NOT the same route as the bubble-up route for `Insert(A)`.

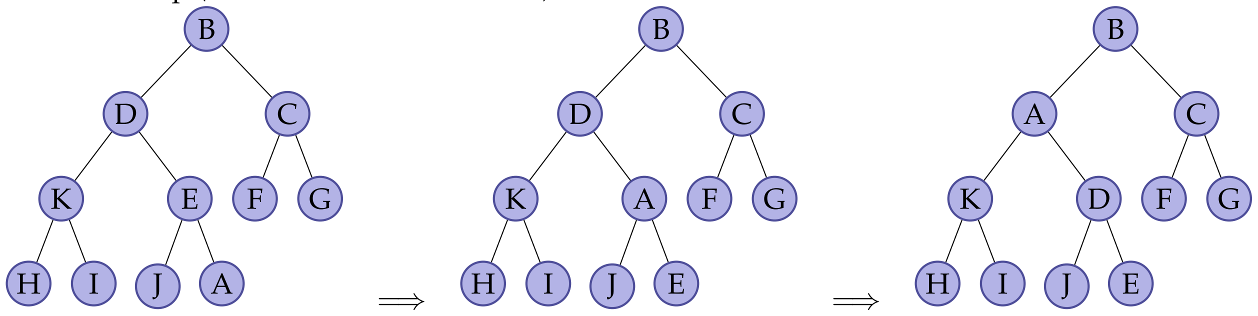
For an arbitrary non-DIS binary max heap,



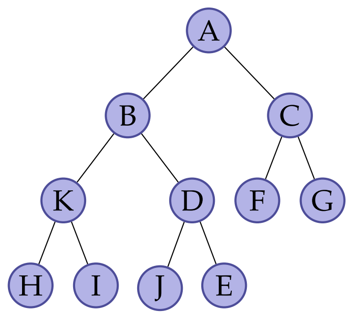
The intermediate heap (after the call to `ExtractMax()`):



The final heap (after the call to `INSERT(A)`):



$\Longrightarrow$


A
B
C
K
D
F
G
H
I
J
E