CS101 Algorithms and Data Structures

Fall 2019

Homework 4

Due date: 23:59, October 20, 2019

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL Name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of above may result in zero score.

Problem 1: Binary Tree & Heap

Binary Tree and Heap have a ton of uses and fun properties. To get you warmed up with them, try working through the following problems.

Multiple Choices: Each question has one or more correct answer(s). Select all the correct answer(s). For each question, you get 0 point if you select one or more wrong answers, but you get 0.5 point if you select a non-empty subset of the correct answers.

Note that you should write you answers of Problem 1 in the table below.

Q(1)	Q(2)	Q(3)	Q(4)	Q(5)
B C D	B	C	B	120, 140, 90, 80, 50, 70, 100, 60, 40

(1) Which of the following statements about the binary tree is true?

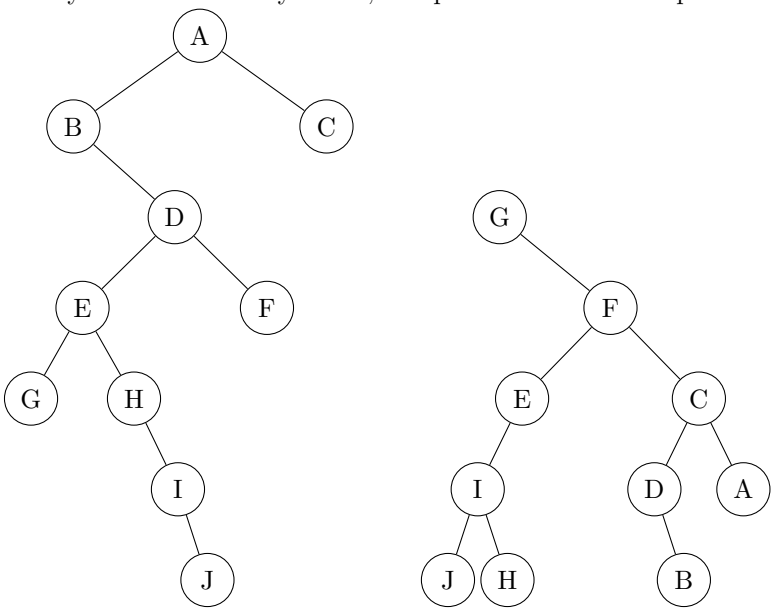
A. Every binary tree has at least one node.

B. Every non-empty tree has exactly one root node.

C. Every node has at most two children.

D. Every non-root node has exactly one parent.

(2) Which traversals of binary tree 1 and binary tree 2, will produce the same sequence node name?



A. Postorder, Postorder

B. Postorder, Inorder

C. Inorder, Inorder

D. Preorder, Preorder

(3) Which of the following statements about the binary tree is **not** true?

A. A rooted binary tree has the property that $n_0 = n_2 + 1$, where n_i denotes the number of nodes with i degrees.

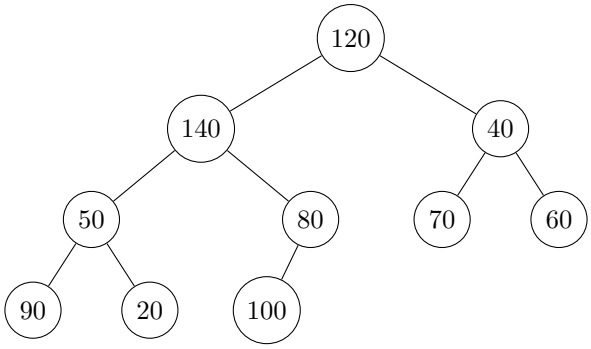
B. Post-order traverse can give the same output sequence as a BFS.

C. BFS and DFS on a binary tree always give different traversal sequences.

D. None of the above.

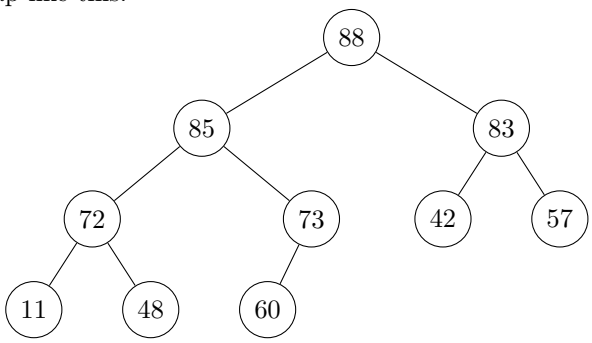
(4) Which of the following statements about the binary heap is **not** true?
A. There exists a heap with seven distinct elements so that the in-order traversal gives the element in sorted order.
B. If item A is an ancestor of item B in a heap (used as a Priority Queue) then it must be the case that the Insert operation for item A occurred before the `Insert` operation for item B.
C. If array A is sorted from smallest to largest then A (excluding A[0]) corresponds to a min-heap.
D. None of the above.

(5) Suppose we construct a min-heap from the following initial heap by Floyd's method. After the construction is completed, we delete the root from the heap. What will be the post-order traversal of the heap? Write down your answer in the table above directly.



Problem 2: Heap Sort

You are given such a max heap like this:



Then you need to use array method to show each step of heap sort in increasing order. Fill in the value in the table below. Notice that the value we have put is the step of each value sorted successfully. For each step, you should always make you heap satisfies the requirement of max heap property.

index	0	1	2	3	4	5	6	7	8	9	10
value		88	85	83	72	73	42	57	11	48	60

Table 1: The original array to represent max heap.

index	0	1	2	3	4	5	6	7	8	9	10
value		85	73	83	72	60	42	57	11	48	88

Table 2: First value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		83	73	57	72	60	42	48	11	85	88

Table 3: Second value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		73	72	57	11	60	42	48	83	85	88

Table 4: Third value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		72	60	57	11	48	42	73	83	85	88

Table 5: Fourth value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		60	48	57	11	42	72	73	83	85	88

Table 6: Fifth value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		57	48	42	11	60	72	73	83	85	88

Table 7: Sixth value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		48	11	42	57	60	72	73	83	85	88

Table 8: Seventh value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		42	11	48	57	60	72	73	83	85	88

Table 9: Eighth value is successfully sorted.

index	0	1	2	3	4	5	6	7	8	9	10
value		11	42	48	57	60	72	73	83	85	88

Table 10: Last 2 values are successfully sorted.

Problem 3: Median Produce 101

Nowadays the hotest variety show *Produce 101* has a new rule to judge all the singers: for all 5 judgers, use the median score value among all the judgers to set her score. Previously, the programme group has a calculator to calculate the score for each singer. Accidentally, the calculator is broken one day. And the fans of famous star Yang Chaoyue are eagerly waiting for the score. So they want to help the programme group to get the correct score.

Recall that the median of a set is the value that separates the higher half of set's values from the set's lower values.

For example, given the set with **odd** numbers of elements:

$$\{78, 94, 17, 87, 65\}$$

The median score is 78.

For another example, given the set with **even** numbers of elements:

$$\{78, 94, 17, 87, 65, 76\}$$

The median score is $(78 + 76)/2 = 77$.

In Yang's fans group, the crazy fans have quarrelled for the following two opinions to get this median score:

- Use only one min heap.
- Use both min heap and max heap.

Now they are asking you for your help. Please help them solve this problem.

So first, let's try the case with only one min heap to get the median score.

Consider a set S of arbitrary and distinct integer scores (not necessatily the set shown above). Let n denote the size of set S, and assume in this whole problem that n can be odd or even. **Assume that we have inserted all the elements in set S to the minheap.**

(1) **Using natural language**, describe how to implement the algorithm that returns the median from set S. Analyze your time complexity. (Suppose the total number of element in S is given, which is n. And the minheap has been built.)

You will receive full credit only if your method runs in $O(n\log n)$ time.

Solution:

1. Firstly, we pop the minimum element of the heap, which is on the root. This leaves a gap at the back of the array, so we can fill the gap with the minimum element.
2. Then we can repeat the process: pop the minimum element and insert it at the end of the array. The number of the repetitions of process 1 is $\lfloor\frac{n+1}{2}\rfloor - 1$(not including process 1) because we only need to get the median 1 or 2 scores, then we can just sort half of the elements.
3. If $n \mod 2 \equiv 0$, output
$$\frac{array[\lfloor\frac{n+1}{2}\rfloor] + array[\lfloor\frac{n+1}{2}\rfloor - 1]}{2}$$
else, output
$$array[\frac{n}{2}]$$

Annotation:

1. The heap is named array here.
2. The pop operation above means:
(1) Pop the top element at index 1;
(2) Move the last entry to the top, whose index is 1;
(3) Compare the current nodes with its two children, and swap with the smallest one, repeatedly.
i. If the current node is smaller than both two children, then the algorithm will be terminated.
ii.If the current node has no children, then the algorithm will be terminated.

The analysis of time complexity:

1. The pop operation: $O(\log n)$.
2. Totally repeating for $\lfloor\frac{n+1}{2}\rfloor$ times.
3. Step 3: $O(1)$.
$\Longrightarrow$ The time complexity is
$$\lfloor\frac{n+1}{2}\rfloor \times \log n + 1 = O(n\log n)$$

And now, let's try the case with both max heap and min heap to get the median score.

Now, another fancy fan Wang Xiaoming finds a data structure for storing a set S of numbers, supporting the following operations:

- `INSERT(x):` Add a new given number x to S
- `MEDIAN():` Return a median of S.

Assume no duplicates are added to S. He proposes that he can use a maxheap A and a minheap B to get the median score easily. These two heaps need to always satisfy the following two properties:

- Every element in A is smaller than every element in B.
- The size of A equals the size of B, or is one less.

To return a median of S, he proposes to return the minimum element of B. **Assume that we have inserted all the elements in set S to the two heaps.**

(2) Using two properties above, argue that this is correct, partially correct or wrong (i.e., that a median is returned). If it is not correct, can we find a strategy that calculates the median in $O(1)$ time complexity? Explain the reason and strategy (if we need) briefly. (Suppose the total number of element in S is given, which is n. And heap A, B have been built.)

Solution:

It is partially correct. When $n \mod 2 \equiv 1$, it is correct. However, when $n \mod 2 \equiv 0$, it is incorrect because we need to find the median two numbers and calculate the average. The strategy is to output the minimum element of B and maximum element of A and calculate the average. Because both of the maximum element of A and the minimum element of B are the root of the heap A and B respectively, the time complexity is $O(1)$ since get the top is $O(1)$.
Notice:

1. If A, B have the same size, pop their roots respectively and calculate the average.
2. If size(A) is one less than size(B), output the root of B.

(3) **Using natural language**, explain how to implement `INSERT(x)` operation. You should notice that these two properties should always be held. Analyze the most efficient running time of `INSERT` algorithm in terms of n. (Suppose the total number of element in S is given, which is n.)

Solution:

1. If x < root of A:

   i. Add x after the last element of A.

   ii. Find the parent of x (the index of the parent is (the index of x) / 2), compare x with its parent.

      1) If the parent of x is smaller than x, swap x with its parent and make the parent of x the new node to compare, returning to the process ii to execute.(It means that the parent of node with value x will replace the x in step ii.)When the node has no parent, stop;

      2) If the parent of x is larger than x, stop.
      The heap A become a max-heap again after the insertion of x.

2. If x >= root of A:

   i. Add x after the last element of B.

   ii. Find the parent of x (the index of the parent is (the index of x) / 2), compare x with its parent.

      1) If the parent of x is larger than x, swap x with its parent and make the parent of x the new node to compare, returning to the process ii to execute.(It means that the parent of node with value x will replace the x in step ii.)When the node has no parent, stop;

      2) If the parent of x is smaller than x, stop.
      The heap B become a min-heap again after the insertion of x.

3. If size(A) > size(B):

   i. Pop the root of A. The pop operation means:
   (1) Pop the top element at index 1;
   (2) Move the last entry to the top, whose index is 1;
   (3) Compare the current nodes with its two children, and swap with the smallest one, repeatedly.

   If the current node is smaller than both two children, then the algorithm will be terminated.

   If the current node has no children, then the algorithm will be terminated.

   ii. Save the initial root of A as b.

   iii. Add b after the last element of B.

   iv. Find the parent of b (the index of the parent is (the index of b) / 2), compare b with its parent.

      1) If the parent of b is larger than b, swap b with its parent and make the parent of b the new node to compare, returning to the process iv to execute.(It means that the parent of node with value b will replace the b in step iv.)When the node has no parent, stop;

      2) If the parent of b is smaller than b, stop.
      The heap B become a min-heap again after the insertion of b.

   v. repeat the step i. to iv. until size(A) = size(B).

4. If size(A) < size(B) −1:

i. Pop the root of B. The pop operation means:
   (1) Pop the top element at index 1;
   (2) Move the last entry to the top, whose index is 1;
   (3) Compare the current nodes with its two children, and swap with the smallest one, repeatedly.

   If the current node is smaller than both two children, then the algorithm will be terminated.

   If the current node has no children, then the algorithm will be terminated.

ii. Save the initial root of B as a.

iii. Add a after the last element of A.

iv. Find the parent of b (the index of the parent is (the index of a) / 2), compare a with its parent.

   1) If the parent of a is smaller than a, swap a with its parent and make the parent of a the new node to compare, returning to the process iv to execute.(It means that the parent of node with value a will replace the a in step iv.)When the node has no parent, stop;
   2) If the parent of a is larger than a, stop.
      The heap A become a max-heap again after the insertion of a.

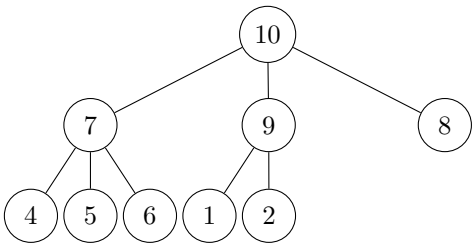
v. repeat the step i. to iv. until size(A) = size(B) -1.

When x is inserted to A and size(A)= size(B), it has the most efficient running time, and the percolation costs $\Theta(n \log_2 n)$.

Problem 4: k-ary Heap

In class, Prof. Zhang has mentioned the method of the array storage of a binary heap. In order to have a better view of heap, we decide to extend the idea of a binary heap to a k-ary heap. In other words, each node in the heap now has at most k children instead of just two, which is stored in a complete binary tree.

For example, the following heap is a 3-ary max-heap.



(1) If you are given the node with index i, what is the index of its parent and its jth child ($1 \leq j \leq k$)?

Notice: We assume the root is kept in $A[1]$. For your final answer, please represent it in terms of k, j and i. Please use flooring or ceiling to ensure that your final answer is as tight as possible if you need, i.e. $\lfloor x \rfloor$, $\lceil x \rceil$.

Solution:

The index of its parent: If $i = 1$, the node has no parent; If $i \neq 1$, the index of its parent is $\lceil \frac{i-1}{k} \rceil$.
The index of its jth child: $ki + j - k + 1$.

(2) What is the height of a k-ary heap of n elements? Please show your steps.

Notice: For your final answer, please represent it in terms of k and n. Please use flooring or ceiling to ensure that your final answer is as tight as possible if you need, i.e. $\lfloor x \rfloor$, $\lceil x \rceil$.

Solution:

In depth i:

$i = 0$, max number of nodes $= k^0 = 1$;

$i = 1$, max number of nodes $= k^1$;

$i = 3$, max number of nodes $= k^2$;

.....

$i = h$, max number of nodes $= k^h$.

Thus, the max number of nodes in a k-ary heap of height h is:

$$maxn = 1 + k^1 + k^2 + ... + k^h = \frac{1 - k^{h+1}}{1 - k}$$

We need to consider that the k-ary heap is not necessarily perfect, so the total number of nodes should be:

$$\frac{1 - k^h}{1 - k} + 1 \leqslant n \leqslant \frac{1 - k^{h+1}}{1 - k} (k \geqslant 2)$$

Solve the inequation, we can get:

$$h \geqslant log_k \frac{1 - n(1 - k)}{k}$$

We can transform the result above to:

$$h \geqslant \frac{ln[1 - n(1 - k)] - 1}{lnk}$$

Hence, The height of a k-ary heap of n elements is

$$\lceil \frac{ln[1 - n(1 - k)] - 1}{lnk} \rceil (k \geqslant 2)$$

(3) Now we want to study which value of k can minimize the comparison complexity of heapsort. For heapsort, given a built heap, the worst-case number of comparisons is $\Theta(nhk)$, where $h = \Theta(\log_k n)$ is the height of the heap. Suppose the worst-case number of comparisons is $T(n, k)$. You need to do:

- Explain why $T(n, k) = \Theta(nhk)$.
- Suppose n is fixed, solve for k so that $T(n, k)$ is minimized.

Notice: k is an integer actually. In this problem, we only consider the complexity of comparision, not the accurate number of comparision.

Solution:

1. In the worst case, for every node, let's see it as a parent node, its k children need $k-1$ comparisons to get the largest or smallest one, and the selected one needs an extra comparison with the parent node so that the total comparisons of this parent node is k. For every depth below this parent node, we need similar comparisons recursively. The total number of comparisons is $\Theta(hk)$. $\Longrightarrow T(1, k) = \Theta(hk)$. We have n nodes in total, then $\Longrightarrow T(n, k) = \Theta(nhk)$.

$$T(n,k) = k(\frac{n}{2} \times h + \frac{n}{4} \times (h-1) + \frac{n}{8} \times (h-2) + ... + \frac{n}{2^h})$$

$$2T(n,k) = k(n \times h + \frac{n}{2} \times (h-1) + \frac{n}{4} \times (h-2) + ... + \frac{n}{2^{h-1}})$$

$$\Longrightarrow T(n,k) = k(nh - (\frac{1}{2}n + \frac{1}{4}n + ... + \frac{1}{2^{h-1}}n + \frac{1}{2^h}n))$$

$$\Longrightarrow T(n,k) = k(nh - 1 + \frac{1}{2^h}n) = \Theta(nhk)$$

2.

$$T(n,k) = \Theta(nhk) \Longrightarrow T(n,k) = anhk + b$$

$$h = \Theta(\log_k n) \Longrightarrow h = c\log_k n + d$$

Then we can get:

$$T(n,k) = anhk + b = an(c\log_k n + d)k + b = anck\log_k n + andk + b(k >= 2, k \in Z^+)$$

$$F(k) = nhk = n\log_k nk = n\frac{k}{\log_n k}$$

$$\frac{dF(k)}{dk} = n\frac{log_n k - \frac{1}{lnn}}{(\log_n k)^2}$$

Let $\frac{dF(k)}{dk} = 0$, we can get:

$$k = e$$

For $k \in Z^+$,we can take $k = 3$.
Hence,

$$k = 3$$

.

(4) TA Yuan is motivated by professor, and he has a new idea. He wants to use k-ary heap to implement the heapsort algorithm. Because he wants to loaf on the job, he chooses $k = 1$. And he argues that we only need to do the `BUILD-HEAP` operation in the heapsort algorithm if $k = 1$. Since we know from the lecture that `BUILD-HEAP` takes $O(n)$ time, he thinks it can actually sort in $O(n)$ time!

From the above, we can conclude his two statements:

- When $k = 1$, the only operation required in heapsort algorithm is `BUILD-HEAP`.
- When $k = 1$, the `BUILD-HEAP` operation will run in $O(n)$ time.

Now you are the student of TA Yuan, and you need to judge his two statements are true or false **respectively**.

- If his statement is true, please explain the reason and prove its correctness.
- If his statement is false, please help him find the fallacy in his argument with your own reason. In the heapsort he proposes, what sorting algorithm is actually performed? What is the worst running time of such a sorting algorithm?

Solution:

1. Statement 1 is true. After the BUILD-HEAP operation, the heap will become a min-heap, which means that every node is smaller than its children. When $k = 1$, every non-leaf node has only one child, so the heap will has been sorted after the BUILD-HEAP operation. Hence when $k = 1$, the only operatin required in heapsort algorithm is BUILD-HEAP.
2. Statement 2 is false. When we are building the heap, we also need to do some percolations. Then the operation will run out of $O(n)$ times but actually in $O(n^2)$ time.
   Actually, it performed selection sort. The worst running time of it is $\frac{n(n-1)}{2}$(The complexity is $O(n^2)$).