CS101 Algorithms and Data Structures
Fall 2023
Homework 6

Due date: November 19, 2023, at 23:59

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 problems correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero points.

1. (12 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 1 point if you select a non-empty subset of the correct answers.

Write your answers in the following table.

(a)	(b)	(c)	(d)	(e)	(f)

(a) (2') Which of the following statements about heaps are true?

A. We can build a heap from an array in linear time through Floyd's method, so we can sort an array in linear time through heap sort.

B. In a complete binary min-heap, the sum of values of the internal nodes are no greater than that of leaf nodes.

C. For any node a in a heap, the subtree of the tree with root a is still a heap.

D. If we use a min-heap to do heap sort, we can sort an array in decreasing order.

(b) (2') Which of the following statements about Huffman Coding Algorithm are true?

A. Huffman Coding Algorithm is a compression method without information loss.

B. The Huffman Coding Tree is a complete binary tree.

C. If character a has a higher frequency than b, then the encoded a has a length no longer than encoded b.

D. Given the set of characters and the **order** of their frequencies but the exact frequencies unknown, we can still determine the length of each encoded character.

(c) (2') Which of the followings can be a set of decoded characters in Huffman Coding Algorithm?

A. $\{00, 0100, 0101, 011, 10, 11\}$

B. $\{0, 100, 101, 10, 11\}$

C. $\{0, 10, 110, 1110, 11110, 111110\}$

D. $\{00, 010, 011, 110, 111\}$

(d) (2') Suppose there are two arrays: $\{a_i\}_{i=1}^n$ is an **ascending** array with n distinct elements. $\{b_i\}_{i=1}^n$ is the reverse of a, i.e. $b_i = a_{n-i+1}$. Which of the following statements are true?

A. If we run Floyd's method to build a min-heap for each of the two arrays, the resulting heap will be the same.

B. If we build a complete binary min-heap by inserting a_i sequentially into an empty heap, the runtime of this process is $\Theta(n)$.

C. If we build a complete binary min-heap by inserting b_i sequentially into an empty heap, the runtime of this process is $\Theta(n)$.

D. If we run heap sort on $\{b_i\}_{i=1}^n$, the runtime is $\Theta(n)$.

(e) (2') Which of the following statements about BST are true?

A. The post-order traversal of a BST is an array of descending order.

B. For a BST, the newly inserted node will always be a leaf node.

C. Given an array, suppose we construct a BST (without balancing) by sequentially inserting the elements of the array into an empty BST. Then the time complexity of this process is $O(n \log n)$ in all cases.

D. Given a BST with member variable `tree_size` (the number of descendants of this node) and a number x, we can find out how many elements in the BST is less than x in $O(h)$ time where h is the height of the BST.

(f) (2') Which of the following statements about BST are true?

A. In a BST, the nodes in a subtree appear contiguously in the in-order traversal sequence of the BST.

B. If we erase a node that has two children, then it will be replaced by the maximum object in its right subtree.

C. There are 5 different BSTs of a set with 3 distinct numbers.

D. If a node doesn't have a right subtree, then its **next** (defined in lectures) object is the largest object (if any) that exists in the path from the node to the root.

2. (6 points) Huffman Coding

After you compress a text file using Huffman Coding Algorithm, you accidentally spilled some ink on it and you found that one word becomes unrecognizable. Now, you need to recover that word given the following information:

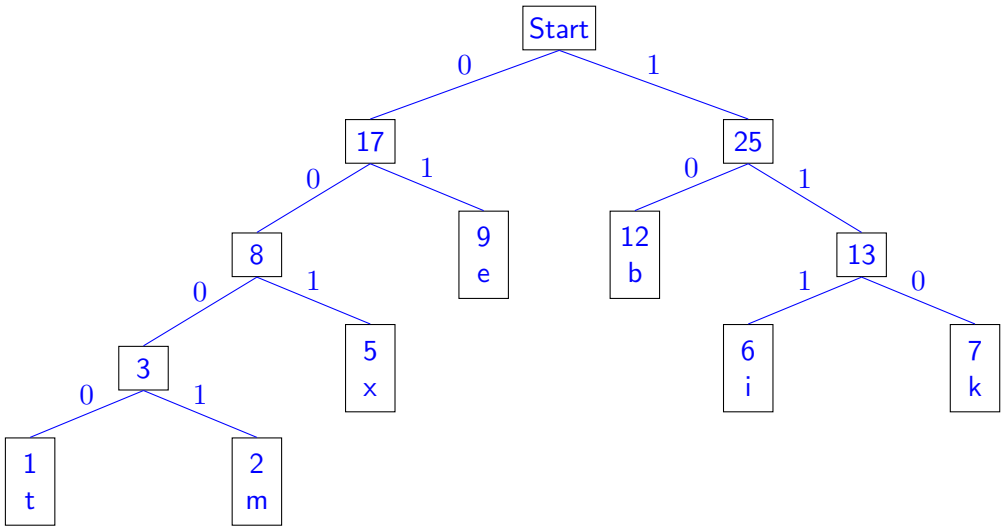
Huffman-Encoded sequence of that word:
00001100111101
Frequency table that stores the frequency of some characters:

characters	b	e	i	k	m	t	x
frequency	12	9	6	7	2	1	5

(a) (4') Please construct the binary Huffman Coding Tree according to the given frequency table and draw the final tree below.
Note: The initial priority queue is given as below. When popping nodes out of the priority queue, the nodes with the same frequency follows "First In First Out".

t	m	x	i	k	e	b
1	2	5	6	7	9	12

Solution:



(b) (2') Now you can "decompress" the encoded sequence and recover the original word you lost. Please write the original word below.

Solution: tieke

3. (9 points) K-Merge

Recall that in merge sort, we learned how to merge two sorted arrays into one in linear time. In this question, we want to design a function that merge K sorted arrays into one.

For example, here are 3 sorted arrays:

$$1, 5, 9$$
$$2, 3, 6, 7$$
$$4, 6, 7, 9$$

and we want to merge them as:

$$1, 2, 3, 4, 5, 6, 6, 7, 7, 9, 9$$

In the following question, suppose the sum of the lengths of the K arrys is n. Here, assume $K = \omega(1)$ and $\log K = o(\log n)$.

(a) (2') Alice does not merge K arrays at once. Instead, she decides to merge 2 of them each time. If she randomly choose 2 arrays and merges them, what is the time complexity of her algorithm in the **worst case**? You don't need to justify your answer.

Solution: $\Theta(kn)$

(b) (2') Recall that in **worst case** merging two sorted arrays with lengths n_1 and n_2 needs $n_1 + n_2 - 1$ comparisons. Now Alice wants to minimize the worst case number of comparisons in her algorithm. How should she choose the two arrays each time? Which algorithm does this strategy coincides with? **Briefly** give your answer.

Solution:

She should always choose the two shortest arrays to merge.

The reason to do this is that when we merge array a and b into ab, we need $n_a + n_b - 1$ comparisons. If we then merge ab and c into abc, we need $n_{ab} + n_c - 1 = n_a + n_b + n_c - 2$ comparisons. Notice the total number of comparisons is $2n_a + 2n_b + n_c - 3$, which means the earlier the array is merged, its length will be counted more times. Therefore, we should merge the shortest arrays first.

This strategy coincides with Huffman Coding (Huffman Algorithm). This is because Huffman Coding is a greedy algorithm that always chooses the two smallest elements (compared by frequency) to merge.

(c) (2') Bob designs an algorithm that merges the K arrays at once by modifying the merge function in merge sort. Each time, he looks up to the front element in each array and finds the smallest one among them. Then he puts this element at the back of his answer array and pop it from its original array. What is the time complexity of Bob's algorithm? **Briefly** justify your answer.

Solution:
$\Theta(kn)$

Each time, Bob needs to find the smallest element among K arrays. This takes $\Theta(K)$ time. Since there are n elements in total, Bob needs to do this $n-1$ times. Therefore, the time complexity is $\Theta(kn)$.

(d) (3') Now you need to improve Bob's algorithm to a better time complexity. **Briefly** describe your algorithm in natural language and give the complexity of your algorithm. Please focus on how to find the smallest front element in a shorter time.

Solution:

1. Heapify the arrays by their front element using Floyd's method. This takes $\Theta(k)$ time.
2. Pop the front element of the array sotred in root, then sift down by the new front element. This takes $\Theta(\log k)$ time.
3. Repeat step 2 for n times.

Therefore, the time complexity is $\Theta(k + n \log k) = \Theta(n \log k)$.

4. (12 points) BST with Duplicates

In our lecture, we require each BST node stands for a single unique element. However, in this question we are talking about BST with duplicated elements. That is, we need to maintain how many identical elements are in the BST. Now, each node may stands for multiple elements with the same value, and we call it the `count` of a node. Here we assume the value is `int` type.

First we give the definition of our `Node` struct.

```
struct Node {
        int val;    // The value of the node.
        int sumCount;    // The number of all elements in the sub-tree
        Node *left, *right; //the left and right child.
    };
```

Note that `sumCount` stands for the number of all elements in the **entire sub-tree**, i.e. the sum of `count` in the entire sub-tree. You will see why we use this definition in the following questions.

For example, if root r has two children a and b, and neither a nor b has a child. Suppose r.`count` $= n_r$, a.`count` $= n_a$ and b.`count` $= n_b$. Then r.`sumCount` $= n_r + n_a + n_b$, a.`sumCount` $= n_a$, and b.`sumCount` $= n_b$.

(a) (3') After figuring out the definition of member variable `sumCount`, you need to design a function that calculates the `count` of a node, using `sumCount` variable. Please complete this function below, and make sure you do not access `nullptr`.

```
// return how many elements a single node stands for
int get_count(Node *a){
        if(a == nullptr)
            return 0;
        int ans = /* (1) */ ;
        if(a->left != nullptr)
            /* (2) */ ;
        if(a->right != nullptr)
            /* (3) */ ;
        return ans;
    }
```

Solution:

```
int get_count(Node *a){
        if(a == nullptr)
            return 0;
        int ans = a.sumCount;
        if(a->left != nullptr)
            ans -= a->left.sumCount;
        if(a->right != nullptr)
            ans -= a->right.sumCount;
        return ans;
    }
```

(b) (4') Given a value v, we want to figure out how many elements are no more than v. Complete the function below, and make sure you do not access `nullptr`. You may use `get_count` function if needed. By calling `count_no_more_than(root, v)`, we can get the number of such elements in the entire BST.

```
// return how many elements are no more than v.
int count_no_more_than(Node *a, int v){
        if(a == nullptr)
            return 0;
        int ans = 0, tmp = 0;
        if (a->left != nullptr)
            tmp = /* (1) */ ;
        if(v < a->value)
            ans = /* (2) */ ;
        if(v == a->value)
            ans = /* (3) */ ;
        if(v > a->value)
            ans = /* (4) */ ;
        return ans;
    }
```

Solution:

```
// return how many elements are no more than v.
int count_no_more_than(Node *a, int v){
        if(a == nullptr)
            return 0;
        int ans = 0, tmp = 0;
        if (a->left != nullptr)
            tmp = a->left.sumCount;
        if(v < a->value)
            ans = count_no_more_than(a->left, v);
        if(v == a->value)
            ans = tmp + get_count(a);
        if(v > a->value)
            ans = tmp + get_count(a) + count_no_more_than(a
                ->right, v);
        return ans;
    }
```

(c) (2') Now suppose you have finished the following two functions correctly. Given the root node of our BST, and two integers l and r, how do you find out the number of elements in the range $[l, r]$? Use variables `root`, `l`, and `r`, and function `count_no_more_than` to write an expression that computes the desired number.

```
count_in_l_to_r = /* Your code */ ;
```

Solution:

```
count_no_more_than(root, r) - count_no_more_than(root, l);
```

(d) (3') **True or False**

(i) If we insert an element into our BST with duplicates, we may need to modify multiple nodes. √ **True** ◯ False

(ii) If we delete a node in our BST with duplicates, the nodes that we need to modify are the nodes on the path from this node to the root. √ **True** ◯ False

(iii) If we use member variable `count` instead of `sumCount`, we can still run `count_no_more_than` in $O(h)$ time, where h is the height of the tree. ◯ True √ **False**