MPCS 55001 Algorithms Autumn 2023
Homework 4

Zhiwei Cao zhiweic
Collaborators: Student A, Student B

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 String Covering (8 points)

Given strings S and T, we say that S *covers* T if S contains all of the same characters in T, including duplicates.

For example, if $S =$ `abcac` and $T =$ `ccba`, then S covers T, but T does not cover S. If $U =$ `bcbc`, then S does not cover U and U does not cover S.

(a) (4 points) Given two strings $A[1..n]$ and $B[1..m]$, compute the first index k such that $A[1..k]$ covers B, or NIL if such an index does not exist. Assume B can have duplicates, and that $n \geq m$ (otherwise there is obviously no solution). Write **pseudocode** for your algorithm, which should run in $O(n)$ **expected** time.

Algorithm 1 String Covering

```
function CoverIndex(A[1..n], B[1..m])
Input: Two strings A[1..n] and B[1..m]
Output: the first index k such that A[1..k] covers B
hashB ← an empty hashmap
for char in B do
    hashB[char] = hashB[char] + 1 //value as the key and number of existence as value
end for
hashA ← an empty hashmap
for j ← 0 to A.length do
    if A[j] in hashB then
        hashB[A[j]] ← hashB[A[j]] − 1
        if hashB[A[j]] ← 0 then
            remove hashB[A[j]]
        end if
        if hashB.size ← 0 then
            return j //index Found
        end if
    end if
end for
return NIL
```

(b) (4 points) Given two strings $A[1..n]$ and $B[1..m]$, return the shortest contiguous substring $A[i..j]$ such that $A[i..j]$ covers B. If there is no such substring, return NIL.

Note that this can be solved in $O(n^2)$ expected time via repeated application of the algorithm in part (a) (taking $i = 1$, $i = 2$, and so on). Write **pseudocode** for an algorithm which runs in $O(n)$ expected time.

Algorithm 2 String Covering B

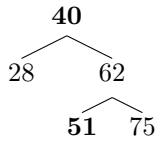
```
function ShoestestCoverString(A[1..n], B[1..m])
Input: Two strings A[1..n] and B[1..m]
Output: return the shortest contiguous substring A[i..j] such that A[i..j] covers B
checkMap ← an empty hashmap
for char in B do
    checkMap[char] = checkMap[char] + 1 //value as the key and number of existence as value
end for
checkLen ← size of checkMap
i ← 0
res ← [0, ∞] //a array for string boundary to store the return string
for index, value in A do
    if checkMap[value] > 0 then
        checkLen ← checkLen − 1
    end if
    checkMap[value] ← checkMap[value] − 1
    if checkLen == 0 then // which indicate the window already covered all element of B
        while Ture do // increase i, to exclude extra unnecessary element
            temp ← A[i]
            if checkMap[temp] == 0 then
                break
            end if
            checkMap[temp] ← checkMap[temp] + 1
            i ← i + 1
        end while
        if (index − i) < res[1] − res[0] then // record the result
            res ← [i, index]
        end if
        checkMap[A[i]] ← checkMap[A[i]] + 1 // increase the place of i, in order to fulfill the window
        checkLen ← checkLen + 1
        i ← i + 1
    end if
end for
if res[1] > A.length then
    return NIL
else
    return A[res[0] ... res[1] + 1] // find the shortest contiguous sub-string A[i..j]
end if
```

2 Augmented BSTs (9 points)

Many problems call for a modified, or **augmented**, binary search tree. An augmented BST stores some extra information at each node in order to facilitate specific operations (in addition to the key, parent, and left/right children).

For example, at every node v, you might store an attribute called $v.max$ which stores the largest value in the subtree rooted at v (including v itself). This augmentation would allow you to answer queries like "What is the maximum value in the BST?" in $O(1)$ time by simply looking at $T.root.max$. Think about what modifications to the insert and delete operations would be necessary to maintain this augmentation.

Design an augmented BST to support a query of the form `SumBetween(T, x, y)`, which returns the sum of keys in T in the interval $[x, y]$ (including the endpoints, if they exist) in $O(\lg n)$ time. For example, the query `SumBetween(T, 30, 51)` on the below tree will return 91.



There are 2 keys in the tree between 30 and 51 (shown in boldface). Their sum is 91.

You will need to decide what extra information to store, modify the insert and delete operations to maintain the augmentation, and write pseudocode for the new operation. The insert and delete operations should still run in $O(\log n)$ time, where n is the number of elements in the tree. You may assume that the BST is balanced, and that the elements in the BST are distinct.

- (1 point) State which additional attribute(s) your augmented BST will maintain.
  The additional attribute I will maintain in the augmented BST is $v.sum$, which will store the sum value of the subtree rooted at v (including v itself)
- (2 points) Write (modified) **pseudocode** for `Insert(T, x)`. Modify the algorithm from CLRS on page 294.

Algorithm 3 Insertion

function Tree-Insert(T, z)
Input: A binary Tree T, and a node z
Output: It modifies T and some of the attributes of in such a way that it inserts into an appropriate position in the tree.
$y \leftarrow NIL$
x = $T.root$
while $x \neq NIL$ **do**
    $y = x$
    $y.sum = y.sum + z.key$
    if $z.key < x.key$ **then**
        $x = x.left$
    else
        $x = x.right$
    end if
end while
$z.p = y$
// z.sum = z.key
if $y == NIL$ **then**
    $T.root = z$
else if $z.key < y.key$ **then**
    $y.left = z$
else
    $y.right = z$
end if
// $z.sum = z.key$

- (3 points) Write (modified) **pseudocode** for `Delete(T, x)`. Modify the algorithm from CLRS on page 298.
- (3 points) Write **pseudocode** for `SumBetween(T, x, y)`. It should run in $O(\lg n)$ time and output an integer.

3 Dynamic Percentiles (12 points)

You are given access to a stream of numeric values of unknown length, e.g., $99, -22.4, \ldots$. The values are not necessarily unique.

(a) (6 points) Design a data structure D which supports the following operations:

Algorithm 4 Deletion

function Tree-Delete(T, z)
Input: A binary Tree T, and a node z
Output: It modifies T and some of the attributes of in such a way that it deletes z an appropriate position in the tree.
if $z.left == NIL$ **then**
    $Transplant(T, z, z.right)$
else if $z.right == NIL$ **then**
    $Transplant(T, z, z.left)$
else
    $y = Tree - Minimum(z.right)$
    if $y.p \neq z$ **then**
        $Transplant(T, y, y.right)$
        $y.right = z.right$
        $y.right.p = y$
        $y.sum = y.sum + (y.right.sum - y.key)$
    end if
    $Transplant(T, z, y)$
    $y.left = z.left$
    $y.left.p = y$
    $y.sum = y.sum + y.left.sum$
end if

Algorithm 5 Sum Between Two Nodes

function SumBetween(T, x, y)
Input: A binary Tree T, and a node x and y
Output: returns the sum of keys in T in the interval [x, y] (including the endpoints, if they exist)
$node = T.root$
if $node == NIL$ **then**
    return 0 // there is no BST
end if
$resSum = 0$
if $x \leq node.key$ and $node.key \leq y$ **then** // node in the range
    $resSum = resSum + node.key$
end if
if $x \leq nopde.key$ **then** // out of the y (right) boundary
    $resSum = resSum + SumBetween(node.left, x, y)$
end if
if $node.key \leq y$ **then** // out of the x (left) boundary
    $resSum = resSum + SumBetween(node.right, x, y)$
end if
return $resSum$

- `D.ingest(x)` stores the next value in the stream in the data structure. This should run in $O(\log n)$ time, where n is the number of values added thus far.
- `D.median()` returns the **true median** of the values added to the data structure thus far. This should run in $O(1)$ time. The **true median** of a sorted sequence $a_1, \ldots, a_n$ is $a_{(n+1)/2}$ if n is odd and $(a_{n/2} + a_{(n/2)+1})/2$ if n is even.

Give a brief explanation of the data structure D. Write **pseudocode** for both operations and justify their running time.
It could be solved by two heap structures: large-Heap and small-Heap. Both heaps have the function of MinHeap (which the root the largest value)
large-heap is used to store the right half of the "sorted" value in the stream, AKA larger medium values. It could ensure that the root value is the smallest in the right half in the stream. All values in the small-heap are larger than medium number
On the contrary, small-heap stores the left half of the "sorted" value in the stream, AKA smaller medium values. It could ensure that the root value is the largest in the left half in the stream. All values in the small-heap are smaller than medium number
The *ingest*() of both heaps takes $O(\log n)$ times since the *insert*() of heap takes $O(\log n)$, and the *median*() only takes $O(1)$, since it just needs to take both root value and calculate the medium value.

Algorithm 6 3A: Ingest values of a Data Stream

function Ingest(x)
Input: A value x in a data stream
Output: Store the values of the stream into different heap structures
// $largeHeap \leftarrow aHeap$, the root is the larger medium value
// $smallHeap \leftarrow aHeap$, the root is the smaller medium value
$smallHeap.insert(-1 * x)$
if smallHeap is not Empty and largeHeap is not empty and $-1 * smallHeap[0] > largeHeap[0]$ **then**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end if
if $smallHeap.length > largeHeap.length + 1$ **then**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end if
if $smallHeap.length + 1 < largeHeap.length$ **then**
    $smallHeap.insert(-1 * largeHeap.ExtractMin)$
end if

Algorithm 7 Find the median of the Stream

function Median()
Input: Two heap structures
Output: returns the **true median** of the values added to the data structure thus far.
if $smallHeap.length > largeHeap.length$ **then** // odd
    return $-1 * smallHeap[0]$
end if
if $smallHeap.length < largeHeap.length$ **then** // odd
    return $largeHeap[0]$
end if
return $(-1 * smallheap[0] + largeHeap[0])/2$

(b) (3 points) Modify your data structure from part (a) to support computing an arbitrary percentile p. For example, if $p = 90$, your data structure should be able to compute the first value greater than or equal to the 90th percentile of the values it has ingested. The value of p is provided as a parameter during initialization, i.e., `D.init(p)`. The running time of the `ingest(x)` operation should be the same as in part (a), and the running time of the new `D.percentile()` operation should be the same as `median()` in part (a).
Set a boundary to manage the size of two heaps based on the p value.
Increase the size of smallHeap based on the p value

The *ingest*() still takes $O(\log n)$ based on the structure of the min heap
The *percentile*() takes $O(1)$ since it just returns the value of min heap with the index 0

Algorithm 8 3B: Ingest values of a Data Stream

function Ingest(x)
Input: A value x in a data stream
Output: Store the values of the stream into different heap structures
// $largeHeap \leftarrow aHeap$, the root is the larger value
// $smallHeap \leftarrow aHeap$, the root is the smaller value
// $p \leftarrow D.init(p)$ an arbitrary percentile p
$smallHeap.insert(-1 * x)$
if smallHeap is not Empty and largeHeap is not empty and $-1 * smallHeap[0] > largeHeap[0]$ **then**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end if
$percentileBoundary \leftarrow \lceil (p/100) * (smallHeap.length + largeHeap.length) \rceil$
while $smallHeap.length < percentileBoundary$ **do**
    $smallHeap.insert(-1 * largeHeap.ExtractMin)$
end while
while $smallHeap.length > percentileBoundary$ **do**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end while

Algorithm 9 Find the median of the Stream

function Median()
Input: Two heap structures and an arbitrary percentile p
Output: returns the first value greater than or equal to the 90th percentile of the values it has ingested.
return $-1 * smallheap[0]$

(c) (3 points) Finally, design an enhanced data structure S which supports the `median` operation over a sliding window of size k. The window size k is provided as a parameter during initialization. You may add an auxiliary data structure if you wish. Describe how you would modify `ingest` and `median`.

For example, suppose $k = 3$ and the stream emits $10, 85, 44, 15$, which are ingested by the data structure. If the `median` operation is called after these four values are processed, it should return 44, as that is the median of the final 3 values. If the stream emits 12 and `median` is called again, it should return 15.

For this data structure, the `ingest` operation should run in $O(\log k)$ time. The `median` operation should still run in $O(1)$ time.

4 Programming: Huffman Codes (20 points)

(a) (15 points) Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; you do not need to include it here.

(b) In addition, there are a few short questions to answer here on LaTeX, after you finish your code.

Consider an alphabet with letters a–h and the following counts in a message.

a	b	c	d	e	f	g	h
24	5	2	7	31	15	9	7

Table 1: Letter counts

(i) (1 point) If every letter is encoded with a fixed length code of three bits, what is the number of bits needed to encode the entire message?
Count of letters = 24 + 5 + 2 + 7 + 31 + 15 + 9 + 7 = 100
Sum bits = Count of letter * 3 bits = 300 bits

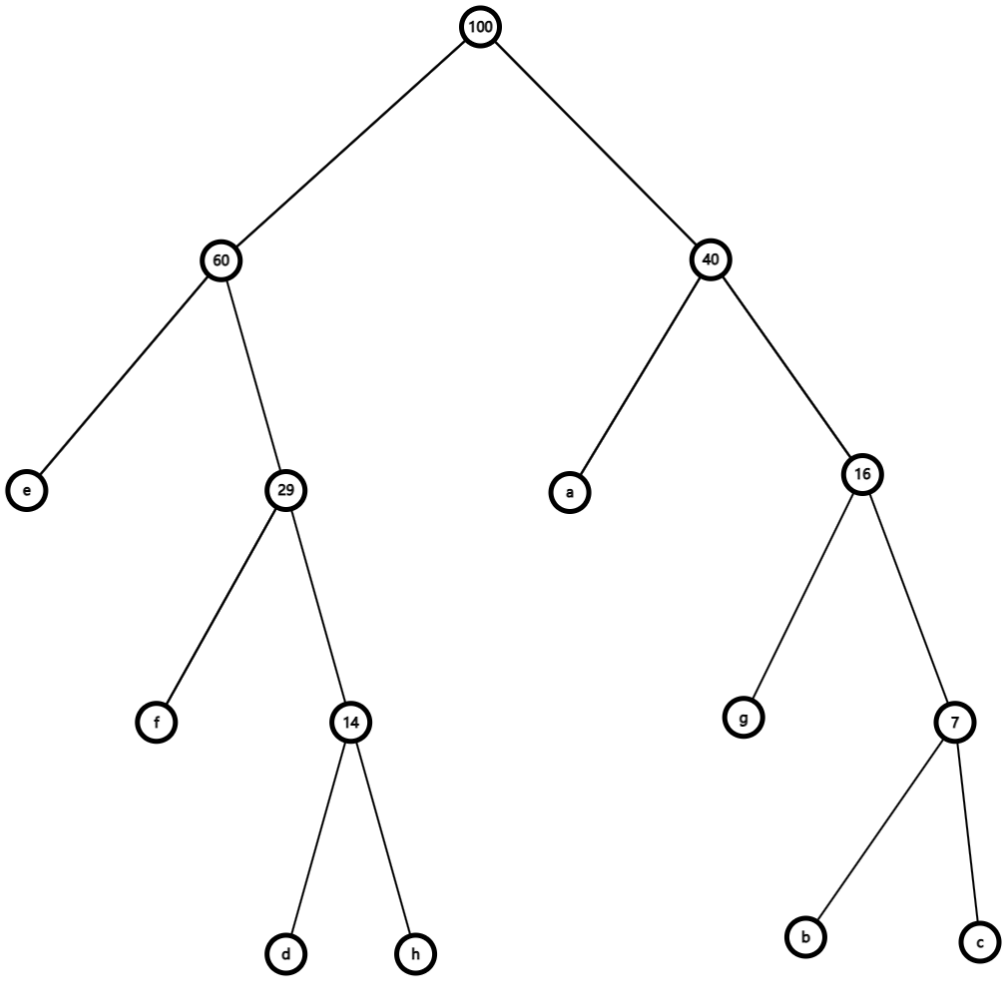
Algorithm 10 3C: Ingest values of a Data Stream

function Ingest(x)
Input: A value x in a data stream
Output: Store the values of the stream into different heap structures
// $largeHeap \leftarrow aHeap$, the root is the larger medium value
// $smallHeap \leftarrow aHeap$, the root is the smaller medium value
// $k \leftarrow$ a sliding window of size k
// $windowQueue \leftarrow aqueue$, in order to store the values of the stream into the window can slide the window. Append() and pop() only take $O(1)$
$smallHeap.insert(-1 * x)$
$windowQueue.append(x)$
if smallHeap is not Empty and largeHeap is not empty and $-1 * smallHeap[0] > largeHeap[0]$ **then**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end if
if $smallHeap.length > largeHeap.length + 1$ **then**
    $largeHeap.insert(-1 * smallHeap.ExtractMin)$
end if
if $smallHeap.length + 1 < largeHeap.length$ **then**
    $smallHeap.insert(-1 * largeHeap.ExtractMin)$
end if
if $windowQueue.length > k$ **then**
    $oldest = windowQueue.pop(0)$
    if $oldest < -1 * smallHeap[0]$ **then**
        $smallHeap.delete(-1 * oldest)$ // takes $O(\log n)$
    else
        $largeHeap.delete(oldest)$ // takes $O(\log n)$
    end if
end if

Algorithm 11 Find the median of the Stream

function Median()
Input: Two heap structures
Output: returns the **true median** of the values added to the data structure thus far.
if $smallHeap.length > largeHeap.length$ **then** // odd
    return $-1 * smallHeap[0]$
end if
if $smallHeap.length < largeHeap.length$ **then** // odd
    return $largeHeap[0]$
end if
return $(-1 * smallheap[0] + largeHeap[0])/2$

(ii) (1 point) Compute and draw the optimal Huffman code tree.



(iii) (1 point) What is the number of bits needed to encode the entire message using the Huffman code?
24 * 2 + 5 * 4 + 2 * 4 + 7 * 4 + 31 * 2 + 15 * 3 + 9 * 3 + 7 * 4 = 266 bits

Char	a	b	c	d	e	f	g	h
Freq	24	5	2	7	31	15	9	7
Code	10	1110	1111	0111	00	010	110	0110
Bits	2	4	4	4	2	3	3	4

Table 2: Letter counts

(iv) (2 points) Write pseudocode for an algorithm that takes a Huffman tree and outputs the letters in descending order of frequency. It should run $O(k)$ time, where k is the size of the alphabet.

Algorithm 12 Letters in descending order of frequency

function getDescendingOrde(huffmanTree)
Input: A Huffman tree
Output: the letters in descending order of frequency
$letterList \leftarrow$ a list to store the letters
$collectLeaveNode(huffmanTree)$ // a method could find all leaves node in the tree
def $collectLeaveNode(curr_node)$
if $curr_node.left == NILandcurr_node.right == NIL$ **then**
    $letterList.append(curr_node)$
    return
end if
$collectLeaveNode(curr_node.left)$
$collectLeaveNode(curr_node.right)$
$letterList.sort()$ // sort the list by the frequency of each letter in a reversed direction
$res \leftarrow$ return list with letters
for i in letterList **do**
    res.append(i.symbol)
end for
return res