CS5050 Advanced Algorithms

Spring Semester, 2021

Assignment 4: Data Structure Design

**Due Date: 11:59 p.m.**, Tuesday, Mar. 23, 2021

1. **(20 points)** Suppose we have a min-heap with *n* distinct keys that are stored in an array *A*[1*... n*] (a min-heap is one that stores the smallest key at its root). Given a value *x* and an integer *k* with 1 ≤ *k* ≤ *n*, design an algorithm to determine whether the *k*-th smallest key in the heap is smaller than *x* (so your answer should be “yes” or “no”). The running time of your algorithm should be *O*(*k*), independent of the size of the heap.

**Remark.** If we were to find the *k*-th smallest key of the heap, denoted by *y*, then the best way would be to perform *k* times *deleteMin* operations, which would take *O*(*k* log*n*) time (or using the selection algorithm, which would take *O*(*n*) time). Our above problem, however, is actually a *decision problem*. Namely, you only need to decide whether *y* is smaller than *x*, and you do not have to know what the exact value of *y* is. Hence, the problem is easier and we are able to solve it in a faster way, i.e., *O*(*k*) time.

**Main Idea:** In order for this algorithm to work in O(k) time we need to start by checking k elements in the heap and while counting the number of elements smaller than x. Then, if we have k elements that are smaller than x, we will return true because the k-th smallest would have to be smaller than x otherwise we would return false. In order for this algorithm to work we only need to check a node’s children if that node is smaller than x.

**Time Complexity:** In the worst case we will examine k + l elements where l represents checking nodes that are larger than x. l will be smaller than k so k + l ends up being O(k) time.

1. **(20 points)** Suppose you are given a binary search tree *T* of *n* nodes (as discussed in class, each node *v* has *v.left*, *v.right*, and *v.key*). We assume that no two keys in *T* are equal. Given a value *x*, the *rank* operation *rank*(*x*) is to return the *rank* of *x* in *T*, which is defined to be one plus the number of keys of *T* smaller than *x*. For example, if *T* has 3 keys smaller than *x*, then *rank*(*x*) = 4. Note that *x* may or may not be a key in *T*. Refer to Figure 1 for more examples.

25

16

48

8

20

5

18

59

23

Figure 1: *rank*(16) = 3, *rank*(21) = 6, *rank*(25) = 7, *rank*(26) = 8.

Let *h* be the height of *T*. We know that *T* can support the ordinary *search, insert*, and *delete* operations, each in *O*(*h*) time. You are asked to augment *T*, such that the *rank* operation, as well as the normal *search, insert*, and *delete* operations, all take *O*(*h*) time each.

You must present: (1) the design of your data structure (i.e., how you augment *T*); (2) the algorithm for implementing the *rank*(*x*) operation (please give the pseudocode); (3) briefly explain why the normal operations *search, insert*, and *delete* can still be performed in *O*(*h*) time each (you do not need to provide the details of these operations).

**Main Idea(1)**: For this algorithm I would augment T by having each node keep track of its size where v.size = v.left.size + v.right.size + 1, with a base case of a leaf node which would have v.size = 1.

**Pseudocode(2)**:

x = node current rank being calculated;

c = current rank which starts as 0;

rank(v, x, c) {

if (v == NULL) {return r};

if (v.key > x) {return rank(v.left, x, r)};

if (v.key < x) {return rank(v.right, x, r + v.left.size + 1)};

if (v.key == x) {return r + v.left.size + 1};

}

**Main Idea(3):** The reason this algorithm does not affect the O(h) time complexity of search, insert, or delete is because the size attribute depends only on v, v.left, and v.right which are contained in the node itself. In class we discussed that doing this would not add to the time complexity of an algorithm.

**Time Complexity:** The algorithm starts at the root of the tree and travels until it reaches a leaf node, meaning it examines at most h tree nodes. This means the worst-case time complexity is O(h).

1

1. **(20 points)** This problem is concerned with **range queries** (we have discussed a similar problem in class) on a binary search tree *T* whose keys are real numbers (no two keys in *T* are equal). Let *h* denote the height of *T*. The range query is a generalization of the ordinary *search* operation. The **range** of a range query on *T* is defined by a pair [*xl,xr*], where *xl* and *xr* are real numbers and *xl* ≤ *xr*. Note that *xl* and *xr* may not be the keys in *T*.

You already know that *T* can support the ordinary *search, insert*, and *delete* operations, each in *O*(*h*) time. You are asked to design an algorithm to efficiently perform the *range queries*. That is, in each range query, you are given a range [*xl,xr*], and your algorithm should report all keys *x* stored in *T* such that *xl* ≤ *x* ≤ *xr*. Your algorithm should run in *O*(*h* + *k*) time, where *k* is the number of keys of *T* in the range [*xl,xr*]. In addition, it is required that all keys in [*xl,xr*] be reported in a *sorted order*. Please give the pseudocode for your algorithm.

**Remark.** Such an algorithm of *O*(*h* + *k*) time is an *output-sensitive* algorithm because the running time (i.e., *O*(*h*+*k*)) is a function of the output size *k*. As an application of the range queries, suppose the keys of *T* are student scores in an exam. A range query like [70*,*80] would report all scores in the range in sorted order.

**Main Idea:** In order for this algorithm to work, we must visit each node of the tree that falls in the range [xl,xr]. Since we are checking each node in that range, this will give us a running time of O(k). The key of each node in the range will be stored in sorted order in a list structure called keys.

**Pseudocode:**

nodesInRange(T, xl, xr) {

a = leastCommonAncestor(T, xl, xr)

if (a == NULL) {return an empty list;}

keys = checkLeft(u.left, xl) keys.append(a.key) keys.append(checkRight(a.right, xr)) {return keys}

}

checkLeft(v, xl) {

(if v == NULL) {return empty;}

if (v.key == xl){

keys.append(v.key)

return keys.append(checkLeft(v.right, xl);}

if (v.key > xl){

keys = checkLeft(v.left,xl)

keys.append(v.key) keys.append(checkLeft(v.right, xl) return keys;}

if (v.key < xl keys){

checkLeft(v.right, xl) return keys};

}

checkRight(v, xr) {

if (v == NULL) {return empty;}

if (v.key) == xr{

keys.append(checkRight(v.left, xr)) keys.append(v.key) ;

return keys;}

if (v.key) < xr{

keys.append(checkRight(v.left, xr)) keys.append(v.key) keys.append(checkRight(v.right, xr))

return keys;}

if (v.key > xr) {

keys = checkRight(v.left, xr)

return keys;}

}

**Time Complexity:** This algorithm takes at the most (h + k) because the size of k is a variable range between [*xl,xr*] and the highest range would include the total height of the tree.

1. **(20 points)** Consider one more operation on the above binary search tree *T* in Problem 3: *range-sum*(*xl,xr*). Given any range [*xl,xr*] with *xl* ≤ *xr*, the operation *range-sum*(*xl,xr*) reports the *sum* of the keys in *T* that are in the range [*xl,xr*].

You are asked to augment the binary search tree *T*, such that the *range-sum*(*xl,xr*) operations, as well as the ordinary *search, insert*, and *delete* operations, all take *O*(*h*) time each, where *h* is the height of *T*.

You must present: (1) the design of your data structure (i.e., how you augment *T*); (2) the algorithm for implementing the *range-sum*(*xl,xr*) operation (please give the pseudocode); (3) briefly explain why the ordinary operations *search, insert*, and *delete* can still be performed in *O*(*h*) time each (you do not need to provide the details of these operations).

**Main Idea(1):** For this algorithm I would augment T by having each node track the sum of its key and of each key below it. I would give each node a data member called v.sum which would equal v.left.sum + v.right.sum + v.key. I would also have a base case where v is a leaf node in which v.sum would be equal to v.key.

**Pseudocode(2):**

rangeSum(T, xl, xr) {

a = lowestCommonAncestor(T, xl, xr)

if (a == null) {return 0}

sum = u.key ;

v = u.left while (v != NULL) {

if (v.key >= xl)

sum += v.key + v.right.sum

if (v.key == xl) {break;}

else {v = v.left}

if (v.key < xl)

v =v.right

v = u.right while (v != NULL) {

if (v.key <= xr)

sum += v.key + v.left.sum

if (v.key == xr) {break;}

else {v = v.right}

if (v.key < xr v = v.left)

}

return sum;

}

**Main Idea(3):** The reason this algorithm does not affect the O(h) time complexity of search, insert, or delete is because the size attribute depends only on v, v.left, and v.right which are contained in the node itself. The theorem states that if an augmentation depends only on v, v.left, and v.right, then the O(h) time complexity of other operations of the binary search tree are not affected.

**Time Complexity:** In the worst case this algorithm would traverse the tree from its root to a leaf node which makes the time complexity O(h).

**Total Points: 80**