CS5050 Advanced Algorithms

Spring Semester, 2018

Homework Solution 4

Haitao Wang

1. The main idea is to start from the root and traverse the heap and count the number of keys that are smaller than x. Specifically, we maintain a count, which is zero initially. If the root is smaller than x, then increase the count by one, and then visit both children of the root. In general, if we are at a node i, if the key of i is smaller than x, then we increase the count by one and then visit both children of i. If the key of i is larger or equal to x, then we do not visit either child of v. The algorithm stops either when no nodes are smaller than x, or when the count is equal to k. After the algorithm stops, if the count is equal to k, then we know that the k-th smallest key is smaller than x; otherwise, the k-th smallest key is larger than or equal to x.

   The pseudocode is given in Algorithm 1. For the running time, since the algorithm will stop one the count is equal to k, the algorithm check at most $2k$ elements of A in the worst case. Thus, the running time is $O(k)$.

Algorithm 1: Decide whether the k-th smallest key of the heap A is smaller than x

Input: A min-heap $A[1, \dots, n]$, a value x, and k
Output: "yes" or "no"

```
Initialize a queue Q = ∅; /* use Q to store the indices of A to be visited */
enquque(Q, 1); /* insert index 1 (i.e., the heap root) into the rear of Q */
count ← 0; /* initialize a count to zero */
while count < k and Q ≠ ∅ do
    i ← dequeue(Q); /* remove the front element of Q and assign it to i */
    if A[i] < x then
        count ← count + 1;
        enquque(Q, 2i); /* insert the left child of A[i] into the rear of Q */
        enquque(Q, 2i + 1); /* insert the right child of A[i] into the rear of Q
         */
    end
end
if count = k then
    return "yes"; /* the k-th smallest key of A is smaller than x */
else
    return "no";
end
```

2. We augment the binary search tree T in the following way. For each node v of T, we associate v with a value $v.size$, which is the number of nodes in the subtree rooted at v.

   If we know the *size* values of the left and right children of v, i.e., $v.left.size$ and $v.right.size$, then we have $v.size = v.left.size + v.right.size + 1$. As discussed in class, this property makes sure that each of the normal *search, insert*, and *delete* operations can still be preformed in $O(h)$ time. Also, we can compute the values $v.size$ for all nodes v of T in $O(n)$ time from the leaves to the root in a bottom-up manner (e.g., by using the post-order traversal).

   The algorithm for $rank(x)$ works as follows. We start from the root and maintain a *count*, which is the number of keys smaller than x that have been found. For each node v, if $v.key = x$, then we increase *count* by $v.left.size$ and finish the algorithm. If $x.key > x$, then we move to $v.left$. If $x.key < x$, then we increase *count* by $v.left.size + 1$. Finally, we return $count + 1$. The pseudocode is given in Algorithm 2. The running time is $O(h)$ since the algorithm only visits the nodes in a path of T from the root to a leaf.

Algorithm 2: rank($T.root, x$)

Input: the root of T and a value x
Output: the rank of x in T

```
v = T.root;
count = 0;
while v ≠ NULL do
    if v.key = x then
        count = count + v.left.size;
        return count + 1;
    end
    if v.key > x then
        v = v.left;
    end
    if v.key < x then
        count = count + v.left.size + 1;
        v = v.right;
    end
end
return count + 1;
```

3. The algorithm for the range query operation is similar to the algorithm for the range-min operation we discussed in class, and the difference is that we use the in-order traversal to report all keys in the range $[x_l, x_r]$. Specifically, the algorithm works as follows.

   We first find the lowest common ancestor of x_l and x_r, and we use u to denote this lowest common ancestor (i.e., u is the highest node in T such that its key $u.key$ is in the range $[x_l, r_r]$). As discussed in class, starting from the root of T, we can find u in $O(h)$ time. If x_l is a key of T, then let v_l denote the node of T whose key is x_l; otherwise, let v_l be the new leaf node created for x_l if we were inserting x_l into T. We consider the nodes in order on the path from v_l up to u. For each such node v except u, if $v.key$ is in $[x_l, x_r]$, we output it, and further, we use an in-order traversal procedure to report all keys in the right subtree of v. Next, we go to the parent of v. If $v.key$ is not in $[x_l, x_r]$, then we simply go to the parent of v. This process is done once we arrive at u.

Next, we report the keys in the range $[x_l, x_r]$ on the right subtree of u in a symmetric way. If x_r is a key of T, then let v_r denote the node of T whose key is x_r; otherwise, let v_r be the new leaf node created for x_r if we were inserting x_r into T. We consider the nodes in order on the path from u *down to* v_r. For each such node v, if $v.key$ is not in the range $[x_l, x_r]$, then we simply go down to its left child; otherwise, we first use an in-order traversal procedure to report all keys in the left subtree of v, and then output $v.key$, and finally go down to the right child of v. This process is done once we arrive at v_r.

To see the algorithm runs in $O(h+k)$ time, the total time for finding u, v_l, and v_r is $O(h)$. The total time for checking the nodes (check whether their keys are in the range $[x_l, x_r]$) in the path from v_l to u is $O(h)$ since the number of nodes in the path is no larger than the height of the tree. Similarly, the total time for checking the nodes in the path from u to v_r is also $O(h)$. Finally, the total time for all the in-order traversal procedures is $O(k)$ because all keys reported by the in-order traversal procedures are in the range $[x_l, x_r]$. Therefore, the total running time of the algorithm is $O(h+k)$.

We can easily implement the above algorithm by recursion, which essentially generalizes the in-order traversal algorithm. The pseudocode is given in Algorithm 3 (initially we call range-report($T.root, x_l, x_r$)). You may verify that the pseudocode is consistent with our algorithm described above.

Algorithm 3: range-report(v, x_l, x_r)

if $v == NULL$ **then**
    return;
end
if $v.key < x_l$ **then**
    range-report($v.right, x_l, x_r$);
end
if $v.key > x_r$ **then**
    range-report($v.left, x_l, x_r$);
end
if $x_l \le v.key \le x_r$ **then**
    range-report($v.left, x_l, x_r$);
    output $v.key$;
    range-report($v.right, x_l, x_r$);
end

To see the correctness of the algorithm, the in-order traversal makes sure all keys in the subtrees are reported in ascending order. Also, we always report the keys in $[x_l, x_r]$ in the path from u to v_l before we report the keys in their corresponding right subtrees, and symmetrically, we always report the keys in $[x_l, x_r]$ in the path from u to v_r right after we report the keys in their corresponding left subtrees. Hence, all keys in $[x_l, x_r]$ are reported in ascending order.

4. We augment the binary search tree T in the following way. For each node v of T, we associate v with a value $v.sum$, which is equal to the sum of all keys in the subtree rooted at v.

   If we know the *sum* values of the left and right children of v, i.e., $v.left.sum$ and $v.right.sum$, then we have $v.sum = v.left.sum + v.right.sum + v.key$. As discussed in class, this property makes sure that each of the normal *search, insert*, and *delete* operations can still be preformed in $O(h)$ time. Also, we can compute the values $v.sum$ for all nodes v of T in $O(n)$ time from the leaves to the root in a bottom-up manner.

   The algorithm for implementing the *range-sum*(x_l, x_r) operation is very similar to the range-min operation we discussed in class. Specifically, the algorithm works as follows.

   We first find the lowest common ancestor of x_l and x_r, and we use u to denote this lowest common ancestor (i.e., u is the highest node in T such that its key is in the range $[x_l, r_r]$). Starting from the root of T, we can find u in $O(h)$ time. If x_l is a key of T, then let v_l denote the node of T whose key is x_l; otherwise, let v_l be the new leaf node created for x_l if we were inserting x_l into T.

   The algorithm use a variable *sum* to maintain the accumulated sum of the keys. Initially, $sum = u.key$ since u is in the range. After the algorithm finishes, *sum* will be equal to the sum of the keys of T in the range $[x_l, x_r]$. We consider the nodes on the path from v_l to u. For each node v in the path, if $v.key$ is in $[x_l, x_r]$, we first set $sum = sum + v.key$ and then set $sum = sum + v.right.sum$ (if v has a right child). Next we move down to the left child of v. If $v.key$ is not in $[x_l, x_r]$, then we simply move down to the right child of v. This process is done once we arrive at v_l.

   Similarly, if x_r is a key of T, then let v_r denote the node of T whose key is x_r; otherwise, let v_r be the new leaf node created for x_r if we were inserting x_r into T. We consider the nodes in the path from u to v_r. For each node v in the path, if $v.key$ is in $[x_l, x_r]$, we first set $sum = sum + v.key$ and then set $sum = sum + v.left.sum$ (if v has a left child). Next we move down to the right child of v. If $v.key$ is not in $[x_l, x_r]$, then we simply move down to the left child of v. This process is done once we arrive at v_r.

   The pseudocode is given below in Algorithm 4.

Algorithm 4: range-sum(T, x_l, x_r)

Input: an augmented binary search tree T

Output: the sum of the keys of T in the range $[x_l, x_r]$

```
find the lowest common ancestor u of x_l and x_r by starting from the root in the same way as
  we discussed in class;
if u == NULL then
    return 0;
end
sum = u.key, v = u.left;
while v ≠ NULL do
    if v.key == x_l then
        sum = sum + v.key;
        if v.right ≠ NULL then
            sum = sum + v.right.sum;
        end
        break;
    end
    if v.key > x_l then
        sum = sum + v.key;
        if v.right ≠ NULL then
            sum = sum + v.right.sum;
        end
        v = v.left;
    else
        v = v.right;
    end
end
v = u.right;
while v ≠ NULL do
    if v.key == x_r then
        sum = sum + v.key;
        if v.left ≠ NULL then
            sum = sum + v.left.sum;
        end
        break;
    end
    if v.key < x_r then
        sum = sum + v.key;
        if v.left ≠ NULL then
            sum = sum + v.left.sum;
        end
        v = v.right;
    else
        v = v.left;
    end
end
return sum;
```