CS5050 Advanced Algorithms

Spring Semester, 2018

Assignment 4: Data Structure Design

Philip Nelson

Worked with: Ammon Hepworth, Raul Ramirez, Hailey Maxwell, and Jack Kiefer

1. **(20 points)** 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.

1. Algorithm Description In order to determine if the k^{th} smallest element is less than x, in $O(k)$, the algorithm will count the number of elements less than x until it has found k of them, or there are no more options to explore. Starting with the root, it is placed on a queue. Then, elements are taken off the queue and their children are added if the element is less than x. The count is also incremented if the element is less than x. When an element is greater than x, its children are also greater than x so all can be thrown out. At the end, if the count is equal to k then there are at least k elements less than x, which implies that the k^{th} smallest element is less than x.

2. Pseudocode *kthLessThanX*

```
template <typename T>
bool kthLessThanX(unsigned int k, T x)
{
  std::queue<unsigned int> q;
  auto pos = 0u, ct = 0u, c1 = 0u, c2 = 0u;
  q.push(pos);
  while (ct <= k && q.size() != 0)
  {
    pos = q.front();
    q.pop();
    if (heap[pos] < x)
    {
      ++ct;
      c1 = pos * 2 + 1;
      c2 = pos * 2 + 2;
      if (c1 < heap.size()) q.push(c1);
      if (c2 < heap.size()) q.push(c2);
    }
  }

  if (ct >= k) return true;
  return false;
}
```

3. Time Analysis The algorithm does not need to find the location of the k^{th} smallest element, only count if there are at least k elements less than x. Since less information is required, the running time can be $O(k)$. In the worst case, the algorithm will need to count k elements less than x which will take $O(k)$ time, at which point it can stop because it has proved that the k^{th} smallest element is less than x.

2. **(20 points)** Explain clearly how you augment T and give your algorithm for performing the *rank* operations. You do not need to give the details for other operations (search, insert, delete), but only need to briefly explain why they still take $O(h)$ time after you augment T.

1. Algorithm Description In order to determine the *rank* of x in the tree T in $O(h)$ time, each node in the tree needs to know the size of its sub-tree. Thus, when traversing the tree, when a key, k_s, is smaller than x, all the keys in the left sub-tree of k_s will also be smaller than x. If a node knows the size of its sub-tree, then the algorithm will add one plus the size of the left sub-tree and follow the right tree. When a key, k_l, is larger than x, the algorithm will follow the left child. If a key, k_e, is equal to x, the algorithm is finished, and will return one plus the size of the left sub-tree plus. The algorithm will traverse to the bottom of the tree in the worst case thus making the time complexity $O(h)$.

2. Pseudocode *rank*

```
rank(T, x) {
  if(cur == null)
    return 0;
  if(cur.data < x)
    return rank(cur.right, x) + cur.left.size + 1
  if(cur.data > x)
    return rank(cur.left, x)
  if(cur.data == xr)
    return cur.left.size + 1
}
```

3. Time Analysis Maintaining this extra information does not increase the time complexity because it can be obtained from the left and right children. The size of a node's sub-tree is $v.left.size + v.right.size + 1$. Therefore the time complexity remains $O(h)$ as per the theorem discussed in class.

3. **(20 points)** Design an algorithm to efficiently perform the *range queries*. That is, in each range query, you are given a range $[x_l, x_r]$, and your algorithm should report all keys x stored in T such that $x_l \leq x \leq x_r$. Your algorithm should run in $O(h + k)$ time, where k is the number of keys of T in the range $[x_l, x_r]$. In addition, it is required that all keys in $[x_l, x_r]$ be reported in a *sorted order*.

1. Algorithm Description The algorithm getInRange returns an array of all the elements in T which are in the range $[x_l, x_r]$. It accomplishes this task efficiently in $O(h + k)$ time. It makes use of the *Least Common Ancestor* function discussed in class to obtain the least common ancestor of x_l and x_r. It then uses the lca as a starting point for a modified in order traversal of the sub-tree rooted at lca to report the keys in the range in sorted order.

2. Pseudocode *getInRange*

```
getRange(cur, xl, xr) {
  if(cur == null)
    return []
  if(cur.data < xl)
    return getRange(cur.right, xl, xr)
  if(cur.data >= xl && cur.data <= xr)
    return getRange(cur.right, xl, xr)
      + cur.data
      + getRange(cur.left, xl, xr)
  if(cur.data > xr)
    return getRange(cur.left, xl, xr)
}

array getInRange(T, xl, xr) {
  lca = LCA(T, xl, xr)
  array = getRange(lca, xl, xr)
  return array
}
```

3. Time Analysis The least common ancestor function takes $O(h)$ time. This leaves $O(k)$ to collect the keys in the range $[x_l, x_r]$. The modified in order traversal *getInRange* checks the current element to see if it is outside the range. If so, it knows that "outer" sub-tree can be skipped; "outer" being the sub-tree to the left of an element greater than the range or to the left of an element smaller than the range. When an element is inside the range, the right sub-tree is retrieved, followed by the current's node data and finally the left. Since some of the nodes outside the range must be passed through in order to collect all the keys in the range, the time complexity is $O(k + 2h)$ because in the worst case, the algorithm will have to pass though nodes outside and the range on both sides all the way down the height of the tree.

Combining the *Least Common Ancestor* and *getInRange* algorithms, the time complexity is $O(k + 3h) \Rightarrow O(k + h)$.

4. **(20 points)** Augment the binary search tree T, such that the *range-sum*(x_l, x_r) operations, as well as the normal *search, insert*, and *delete* operations, all take $O(h)$ time each, where h is the height of T.

1. Data Sctructure Description The augmentation used to achieve *range-sum* in $O(h)$ time requires that each node include the sum of its sub-tree along with its data. The extra information is maintained without increasing the complexity of the other operations of the tree because it is obtained from the left and right children. The sum of a node's sub-tree is $v.left.sum + v.right.sum + v.data$. Therefore the time complexity remains unchanged as per the theorem discussed in class.

2. **Pseudocode** *range-sum*

```
sumRangeLeft(cur, xl) {
  if(cur == null)
    return 0
  if(cur.data < xl)
    return sumRangeLeft(cur.left, xl)
  if(cur.data >= xl)
    return sumRangeLeft(cur.left, xl) + cur.data + cur.right.data
}

sumRangeRight(cur, xr) {
  if(cur == null)
    return 0
  if(cur.data > xr)
    return getRangeRight(cur.right, xr)
  if(cur.data <= xr)
    return getRangeRight(cur.right, xr) + cur.data + cur.left.data
}

range-sum(T, xl, xr) {
  lca = LCA(T, xl, xr)
  sum = lca.data + sumRangeRight(lca.right, xr) + sumRangeLeft(lca.left
  return sum
}
```

3. **Time Analysis** The *range-sum* function can compute the sum of the range $[x_l, x_r]$ in $O(h)$ time with the help of the aforementioned augmented binary search tree. Since each node knows the sum of its sub-tree, the algorithm only needs to travel the "outside" paths of the binary tree from the least common ancestor to the bound x_l or x_r. Worst case this is from the root to a leaf of the tree, along both paths, which would be $O(2h)$. As the algorithm follows the "outer" paths, it can make use of the maintained sum in each node's sub-tree removing its need to traverse those sub-trees. This allows the algorithm to execute in $O(h)$ time.

   Combining the *Least Common Ancestor*, *sumRangeLeft*, and *sumRangeRight* algorithms, the time complexity is $O(3h) \Rightarrow O(h)$.

Total Points: 80