Fall 2021: CSOR 4231 - Analysis of Algorithms

Aayush Kumar Verma, UNI: av2955

Homework 3 (Due Date: November 02, 2021)

Problem 1:

Think of the heap in the form of a Binary Tree. As this is a min-heap, the minimum value is always present at the root of the respective sub-trees and the entire tree as a whole. We will follow the property of a heap represented as an array:

$$\textit{Left child} = \textit{heap}[i * 2]$$

$$\textit{Right child} = \textit{heap}[i * 2 + 1]$$

Now, if we traverse the tree level by level expanding their children one-by-one if they satisfy the criteria:

$$\textit{Left child} < x$$

$$\textit{Right child} < x$$

If these conditions return *true* then the nodes are expanded. We will follow a logic similar to Breadth First Search (BFS) to expand the nodes one-by-one and we store the explored nodes in a list/array and return it at the end.

We will make use of a Queue data structure for ease of operation of a FIFO Architecture. The index of the current number (if the above conditions hold true) will be added to the queue and then their children will be processed. Instead of a full-length BFS, we will stop exploring a branch of sub-tree when we find that the next expanded node is not lesser than x. The inputs - heap array: *heap[]*, key: x, and size of heap: n are given to a function *Solve()* which returns a list/array of numbers from *heap[]* which are lesser than x, the size of which will be denoted by k.

Algorithm 1 s-minimum Elements in a Min-Heap

```
Solve (heap [], x, n):
  Create queue
  Initialize result = []
  if heap [1] < x then
      queue.add (1)
  else
      return result
  end if
  while queue.size > 0: do
      root = queue.remove
      result.append (heap [root])
      if (root * 2) < n and heap [root * 2] < x then
          queue.add (root * 2)
      end if
      if (root * 2 + 1) < n and heap [root * 2 + 1] < x then
          queue.add (root * 2 + 1)
      end if
  end while
  return result
```

Time Complexity Analysis and Proof of Correctness:

From the property of a min-heap, we know that a child node if always greater than or equal to it's parent node but never lesser than it. Hence, while exploring the nodes when we find a node whose value is greater than x, we are sure that no child in it's left and right sub-trees will be lesser than x as stated by the above property. Hence, our algorithm runs for only those k nodes which are actually lesser than x giving us a time complexity of $O(k)$. This algorithm also maintains the Shape and Order properties of a Min-heap by exploring the child nodes of a particular parent only and not going beyond $n/2$ thus ensuring that the algorithm only goes up to the leaves of the binary tree, hence it maintains the invariant that an explored node is not null and is always greater than or equal to it's parent. Hence, the algorithm is correct.

Problem 2:

We will run a modified version of the In-order traversal of a Binary Search Tree (BST) for this question. The inputs - root of the tree: *root* and closed interval ranges denoted by x and y, and are given to a function *Inorder()* which finds the numbers which lie in the specified range. We will also maintain a global array/list called *result* which will store the final output of the *Inorder* function. Every *root* will have an integer parameter *val* containing it's value, a pointer to it's left child *left* and a pointer to it's right child *right*. The pointers may be NULL indicating that there is no child.

Algorithm 2 Modified In-Order Traversal of a BST

```
global result = []
Inorder (root, x, y):
  if root is NULL then
      return
  end if
  if root.val > x then
      Inorder (root.left, x, y)
  end if
  if root.val ⩾ x and root.val ⩽ y then
      result.append (root.val)
  end if
  if root.val < y then
      Inorder (root.right, x, y)
  end if
```

Time Complexity Analysis and Proof of Correctness:

Since unique nodes are arranged in a BST in way such that nodes lesser than the parent node are in the left sub-tree of the parent and the greater nodes are in the right sub-tree and the in-order traversal traverses the left sub-tree first, then the parent and then the right sub-tree, thus it ensures that the final result is always in sorted order. Our modified algorithm ensures the following three things (which prove that it works correctly):

1. If the value of the root lies in the specified range, only then is it appended to the output list.
2. The left sub-tree is explored only if the parent node has value greater than x, implying that there may still be some smaller values in the left sub-tree that lie in the specified range.
3. The right sub-tree is explored only if the parent node has value lesser than y, implying that there may still be some greater values in the right sub-tree that lie in the specified range.

These constraints ensure that useless nodes are never visited and the traversal moves from the root to the leaves in a single path of length equal to the height of the tree. Hence, the overall time complexity is given by $O(h + s)$ where s is the number of traversed.

Problem 3:

Let's assume that the given *k* lists are Linked Lists and every node of this list contains two parameters, an integer value *val* of this node and a pointer *next* to the next node. The last node points to NULL. We will maintain a Min-Heap which will work as a Priority Queue (*pq*) data structure where the priority is determined by *node.val*, i.e., lesser the value, higher the priority. The inputs are an array of lists represented by *arr[]* and the number of lists represented by *k*. Initially, we push all the front nodes into the Priority Queue and then using the Min-Heap property, keep extracting the root of the heap (current minimum value) and appending it to our resultant list *result.* As we extract a node, we push it's *next* node into the Priority Queue unless it is NULL. In this way, our resultant merged list will be a single list containing all the nodes in increasing order of value.

Algorithm 3 Merge k-Sorted Lists

```
MergeKSortedLists (arr [], k):
  Create pq of type LinkedListNode and priority defined by node.val
  Initialize dummyhead of type LinkedListNode with any random value
  curr = dummyhead
  for i = 1 to k do
      if arr[i] != NULL then
          pq.add (arr[i])
      end if
  end for
  while pq.size > 0 do
      node = pq.remove
      if node.next != NULL then
          pq.add (node.next)
      end if
      curr.next = node
      curr = curr.next
  end while
  return dummyhead.next
```

Time Complexity Analysis and Proof of Correctness:

At a time, there are at most *k* nodes in the Priority Queue and we are extracting them one-by-one and we know that the extract operation of a heap works in logarithmic time. Here, we have k nodes at a time, hence the time complexity for this operation is *O(log(k)).* Also, we are building the heap with all the elements of all the lists, say *N* nodes, then the overall time complexity becomes *O(Nlog(k)).* The algorithm works correctly because we are maintaining both the Shape and Order properties of the heap by pushing the nodes one at a time, keeping the size fixed at *k* and extracting the node with the minimum value first.

Problem 4:

For this problem, we use Bucket Sort by maintaining n buckets since the numbers range from $[1 \dots n]$ in each list. Each bucket will store a key-value pair (duplicates may be possible) where the key is the number itself and the value is the index of the list in which it belongs. The algorithm works as follows:

1. Traverse each list one-by-one and put the pair of i^{th} number and j^{th} list into the bucket for the i^{th} number.
2. Every bin, at the end of the above iteration, will contain the numbers and their respective list indices.
3. Now, create empty auxiliary lists for each of the input lists. We will use these lists to contain the final output (sorted numbers in each list).
4. Start picking pairs from buckets one-by-one starting from the first bucket in increasing order. Append the number to the auxiliary list with the index obtained from the pair.
5. Return all the lists.

Time Complexity Analysis and Proof of Correction:

We require *O(m + n)* time to place all the numbers in the buckets as we are traversing each list once and accessing each of the m numbers. Then placing them back into the auxiliary lists will take *O(m + n)* time again. Hence, the overall time complexity will be *O(m + n)*.

This algorithm works correctly because we are using a Bucket Sort algorithm which stores the information about both the number and the list where it came from unlike a Counting Sort algorithm in which we would not have been able to store the information about the list indices. Hence, it will be always ensured that the number goes back to the correct list index using Bucket Sort of key-value pairs.

Problem 5:

For this problem, we will use a self - balancing Binary Search Tree (BST) which will store the available space in each node. A self - balancing BST will ensure that the order of the keys is maintained throughout by performing insertion and deletion operations of nodes, when necessary, in *log(n)* time as taught in the class. Every node will be augmented to store which objects lie in that bin. At the end, all the extracted nodes (bins that are full) and the nodes remaining in the BST will contain the mapping of which object went to which bin. The algorithm works as follows:

Best-Fit Algorithm for Bin Packing Problem:

1. Create a tree with an empty bin as the root.
2. Put the first object in the bin and update it's available space (key) to $10 - w_i$, where w_i denotes the weight of the i^{th} object.
3. If the bin is full, extract it from the tree and store in a list.
4. If $w_i >$ available space, traverse the right sub-tree to check if it fits in any node. If no space is available in any bin, create a new bin, put the object in it, and insert this node to the right sub-tree. Update the available size of this node to $10 - w_i$.
5. If $w_i <=$ available space, then find the smallest possible available space (traverse the left sub-tree) and put this object in that bin and update it's weight to $previous\_available\_space - w_i$.

Time Complexity Analysis and Proof of Correctness:

We will be accessing every node once, hence taking *O(n)* time. For putting them into the bins, we are making a choice at every node of the tree, hence traversing only along the height of the tree where the height of the tree is *O(log (number of bins))*, say *k*, so the overall time complexity becomes *O(n log(k))*. We should also note that the number of bins (k) will depend on the input, which in the worst case can be such that every object requires one bin, i.e. a total of n bins giving us the worst-case time complexity of *O(n log(n))*. Self-balancing the tree by insert and delete operations will take *O(log k)* time but this time will be dominated by the time required for the algorithm, hence the overall time complexity remains *O(n log(k))*.

This algorithm works correctly because we are maintaining the Order Property of a BST by updating the available space every time we place an empty bin and removing a bin when it becomes full. Since, the best bin is taken for every object, there may be a case when we need to reorder the nodes to maintain the shape and order of the BST, which is why we are using a self-balancing BST as our data structure which will take care of this whenever an update is made on the nodes of the tree.

Problem 6:

We create a balanced Binary Search Tree (like a Red-Black Tree, AVL Tree, etc.) in which the nodes are ordered based on the birth-dates of the employees. Each node will be augmented to contain additional information about the salary of the employee, the sum of salaries of the employees in the sub-trees, and the size of the sub-trees. Nodes having a duplicate birthday and/or salary will be pushed to the left sub-tree based on the birth-date value. Maintaining a balanced BST ensures that the insert and delete operations take place in *O(log n)* time, as taught in the class lectures. The root node contains information about the birth-date, the salary, size of it's sub-tree, sum of salaries in it's subtrees, and pointers to it's left and right child nodes, which may be null.

(i). *MinBday*: What is the minimum birth-date?

As we know that in a balanced BST, nodes are arranged in the following manner:

1. Child_Key $\leqslant$ Parent_Key $\longrightarrow$ traverse the left sub-tree.
2. Child_Key $>$ Parent_Key $\longrightarrow$ traverse the right sub-tree.

Hence, the minimum birth-date will always be the leftmost node, so we keep going left until the last leftmost leaf node and return it's birth-date.

The time complexity for this operation will be *O(log n)* as we will only be traversing the along height of the tree once where the height is equal to *log (n)*.

(ii). *Count(d)*: How many employees have birth-date $\leqslant$ d?

At every node, we will have 3 choices:

1. *root.birth-date* $= d$: This implies that all the nodes in the left sub-tree will surely have birth-date less than or equal to d and every node in the right sub-tree will have birth-date greater than or equal to d. So, we check whether the left sub-tree is null or not. If not null, we return the size of the left sub-tree + 1 for the current node, else we return 1 for the current node.
2. *root.birth-date* $< d$: If this condition is true, then every node in the left sub-tree will have birth-date less than or equal to d and there may be some more nodes which have birth-date less than or equal to d in the right sub-tree. Hence, we recur for the right sub-tree and return the size of the left sub-tree + 1 (for the current node) + value returned by the right sub-tree.
3. *root.birth-date* $> d$: No more nodes in the right sub-tree will be possible answers as all these nodes will have birth-date greater than d due to the Order Property of a balanced BST. Hence, recur for the left sub-tree only.

The base case is when root is NULL, so we return 0.

The time complexity of this procedure is *O(log n)* we are traversing along the height of the tree, making decisions at every node and traversing only either of the 2 sub-trees of a node.

(iii). *AvgSal(d)*: What is the average salary of employees with birth-date $\leqslant$ d?

We will modify the above stated *Count(d)* routine slightly for this question. Instead of returning the count of birth-dates, we will return the *sum* of the salaries. The recursion logic and the base case remain the same. The time complexity will be *O(log n)* once again as we are traversing only along the height of the tree making choices of sub-trees at every node. We will put this code inside a helper function called *Salary* and the *AvgSal* function will output the average salary by dividing the *sum of salaries* returned by *Salary()*

by *count of such employees* returned by the above defined *Count()* method.

Algorithm 4 Augmented Balanced BST Operations

```
MinBday (root):
  if root is NULL then
      return NULL
  end if
  while root.left is not NULL do
      root = root.left
  end while
  return root.birthdate



Count (root, d):
  if root is NULL then
      return 0
  end if

  if root.birthdate == d then
      if root.left is NULL then
          return 1
      else
          return root.left.size + 1
      end if
  end if

  if root.birthdate > d then
      return Count(root.left, d)
  end if

  if root.birthdate < d then
      if root.left is NULL then
          return 1 + Count(root.right, d)
      else
          return root.left.size + 1 + Count(root.right, d)
      end if
  end if



Salary (root, d):
  if root is NULL then
      return 0
  end if

  if root.birthdate == d then
      if root.left is NULL then
          return root.salary
      else
          return root.left.salary_sum + root.salary
      end if
  end if

  if root.birthdate > d then
      return Salary (root.left, d)
  end if

  if root.birthdate < d then
      if root.left is NULL then
          return root.salary + Salary(root.right, d)
      else
          return root.salary + root.left.salary_sum + Salary(root.right, d)
      end if
  end if
```

```
AvgSal (root, d):
    return Salary (root, d) / Count (root, d)
```

Proof of Correctness:

1. **MinBday:** This is correct because of the order property of a BST, the node with the lowest key is always the leftmost node of the BST.
2. **Count(d):** The algorithm works correctly as it is similar to the one discussed in class and the process is similar to that and the only change is in the return value.
3. **AvgSal(d):** This is also correct because it is similar to the execution of *Count(d)* procedure, which is also correct. Here, we are only returning a different value.