CSC4120 Spring 2024 - Written Homework 4

Yohandi 120040025
Andrew Nathanael 120040007

March 6, 2024

BSTs

Problem 1.

Answer the questions below. Indicate the reason.

(a) We have seen a few different methods to obtain a sorted sequence of integers. Given an array of N elements to be sorted, what are the best- and worst-case runtimes for the following sorting methods? (The time for building a data structure should be considered.)

(i) Insertion Sort

(ii) Merge Sort

(iii) Heap Sort

(iv) In-order traversal of BST

(v) In-order traversal of AVL Tree

(b) You are given a list of N numbers and you like to design a sorting algorithm that uses BSTs. In particular, you construct a corresponding BST by inserting the keys of the list one by one, and then output the nodes using an in-order traversal of the tree. Then the complexity is a) $O(N)$, b) $O(N \log N)$, c) $O(N^2)$.

(c) Suppose that in (b) you use a balanced BST instead. Then the complexity is

(a) $O(N)$

(b) $O(N \log N)$

(c) $O(N^2)$.

(d) We have N keys already stored as a Max-Heap and as an AVL tree. Since both have a maximum depth of $O(\log N)$, searching if k is in the set of keys takes the same time complexity (T/F).

(e) A Max-heap captures the same information as a balanced BST regarding the ordering of the keys of the nodes (T/F).

(f) Outputting the keys in increasing order from a given Min-Heap and a balanced BST take the same time complexity (T/F).

(a) The table below shows both the best-case and the worst-case time complexity for the aforementioned sorting methods.

Sorting Method	**Best-Case**	**Worst-Case**
Insertion Sort	$O(N)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$
Heap Sort	$O(N \log N)$	$O(N \log N)$
In-order traversal of BST	$O(N \log N)$	$O(N^2)$
In-order traversal of AVL Tree	$O(N \log N)$	$O(N \log N)$

(b) The time complexity would be $\mathcal{O}(N^2)$. When the tree is reversely sorted, the insertion on the binary search tree requires $\mathcal{O}(N)$ due to the tree being a line tree. As there are N insertions, it takes $\mathcal{O}(N^2)$ in overall.

(c) Similar to the above; however, this time, each insertion takes $\mathcal{O}(\log N)$ operations, resulting in a $\mathcal{O}(N \log N)$ performance.

(d) AVL tree holds the property of binary search tree, for which, in the worst case scenario, checking the existence of element k requires $\mathcal{O}(h) = \mathcal{O}(\log N)$. However, different things happen for Max-Heap as the data structure does not support key searching in $\mathcal{O}(h)$; in a worst-case scenario, by only utilizing the peek function, finding the existence of k requires popping out all elements in the heap for which it would still be better to iterate of all elements in the heap with $\mathcal{O}(n)$ performance. The statement is false.

(e) The statement is false. Max-heap holds the property of having a node larger than all its descendants' nodes. At the same time, the AVL tree maintains the property of having all left subtree descendants smaller than the current node (and right subtree descendants larger than the current node). Such property allows the AVL tree to perform key searches, unlike max-heap.

(f) The statement is false. By performing in-order traversal on BST, outputting the array's keys in increasing order can be done in $\mathcal{O}(n)$; however, from any min-heap, the process requires $\mathcal{O}(n \log n)$ operations (either by searching using the best first method or popping out the minimum element one by one).

Problem 2.

Build a balanced binary search tree from a sorted list in $O(n)$ time. This time, by balanced binary search tree we mean that its height is $O(\log n)$. Note that, given a list of n numbers, we can build a binary search tree containing these numbers by starting with an empty tree and inserting the numbers from the list one by one into the tree. By employing appropriate rebalancing procedures, e.g. as in AVL-trees, the total time needed to build this tree will be $O(n \log n)$. Now, assume that the elements in this list of n numbers given to you are already sorted. Show how to construct in $O(n)$ time a binary search tree containing these numbers that is balanced.

Given a sorted array, we aim to construct a balanced BST, where the height difference between any node's left and right subtrees is minimized. To achieve this, the algorithm

consistently selects a middle element from the array segment as the root, ensuring that the sizes of the left and right subtrees are equal. This process is recursively applied to all subtrees, effectively balancing the tree. The procedure is as follows:

```
procedure buildBST(t, arr, left, right):
    mid = (left + right) // 2
    t -> value = arr[mid]
    t -> left = new node()
    t -> right = new node()

    build(t -> left, arr, left, mid - 1)
    build(t -> right, arr, mid + 1, right)
```

Note that arr must be passed by reference so that the overall complexity still holds in $\mathcal{O}(n)$. Proof of the complexity can be derived as follows:

$$\begin{aligned} T(n) &= 2T(n/2) + \mathcal{O}(1) \\ &= \mathcal{O}(n) \text{ (by Master's Theorem)} \end{aligned}$$

Problem 3.

Given a binary search tree t, find its i-th smallest element in $O(h)$ time, where h is the height of the tree.

Hint: Use augmentation: for each node x, store the attribute $\text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1$, i.e., the size of the subtree rooted at x.

In a binary search tree, for any given node x, all nodes in the left subtree of x have values less than the value of x, and all nodes in the right subtree of x have values greater than the value of x. This property holds for every node within the tree; thereby, the following algorithm can used to find the i-th smallest element of a binary search tree t.

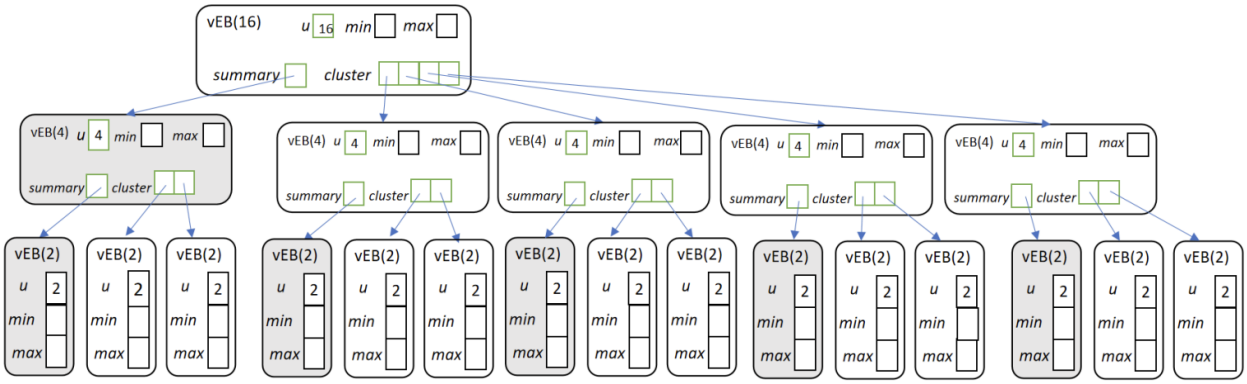
```
procedure ith_smallest(t, i):
    if size[t -> left] + 1 == i:
        return t -> value

    if i <= size[t -> left]:
        return ith_smallest(t -> left, i)
    else:
        return ith_smallest(t -> right, i - size[t -> left] - 1)
```

The answer is obtained by calling ith_smallest(t, i), where t is the root of the binary search tree and i is the index that we are looking for. This works well due to the following considerations:

- If the size of the left subtree is exactly $i-1$, it implies that the current node is the i-th smallest element, as there are precisely $i-1$ elements smaller than the current node.
- If the size of the left subtree is greater than or equal to i, the i-th smallest element must reside within the left subtree. In this case, we continue the search within the left subtree using the same algorithm.
- If the above conditions are unmet, the i-th smallest element is located in the right subtree. The search proceeds in the right subtree for the ($i-$size of the left subtree-1)-th smallest element, effectively finding the i-th smallest element.

vEB trees

Problem 4.



Show the state of the vEB tree that stores $\{0, 2, 5, 6, 11, 12, 15\}$, when $u = 16$.