D3: Qsort, hrúgur, ST, BST

Birgitta Feldís Bjarkadóttir   Edda Steinunn Rúnarsdóttir

September 24, 2017

1. What is the most frequent word in "A tale of two cities" of length at least 7 that starts with the same letter as your first name? How frequent is it?

Solution. *Letters for both our first names were tested and both lower and uppercase letters were taken into an account:*

- ***B**: most frequent word starting with B is **brought** and appears 78 times*
- ***E**: most frequent word starting with E is **English** and appears 30 times*

2. Give the sequence of nodes examined when the methods in BST are used to compute each of the following quantities for the tree

Solution. *answers:*
a. ***EQ*** *(result: Q)*
b. ***EQ*** *(result: Q)*
c. ***EQ*** *(result: Q)*
d. ***EQJ*** *(result: 3)*
e. ***EDQJMT*** *(result: 7)*
f. ***EDQJMT***

3. Suppose that a certain BST has keys that are integers between 1 and 10, and we search for 5. Which sequence below cannot be the sequence of keys examined?
a. 10, 9, 8, 7, 6, 5
b. 4, 10, 8, 6, 5
c. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5
d. 2, 7, 3, 8, 4, 5
e. 1, 2, 10, 4, 8, 5

Solution. ***d. is an impossible sequence for nodes to be examined**, because if the given sequence would hold the node with key 8 would be in the left subtree of the node with key 7 meaning it should be smaller than 7. Node with key 8 can therefore not be in the left subtree of 7 and the sequence is invalid.*

4. Answer problem VI (BST) from final exam from 2015.

Solution. ***Result tree option E.***

5. Suppose that your application will have a huge number of find the maximum operations, but a relatively small number of insert and remove the maximum operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array? Explain.

Solution. *We explored the order of growth time complexity given the three priority queues implementation techniques. See results in below table:*

operation	***heap***	***unordered array***	***ordered array***
find maximum	*O(1)*	*O(N)*	*O(1)*
remove maximum	*O(lg(N))*	*O(N)*	*O(1)*
insert	*O(lg(N))*	*O(1)*	*O(N)*

*Upon exploring table we find that both **heap** and **ordered array** are suitable for this type of priority queue since the find maximum operation is constant for both types. The remove maximum and insert operations are both practically irrelevant to the order of growth so we'd have no real preference between the two types.*

*Although, making the logical assumption that we'd insert more often than or at least as many times as we'd remove from a queue then the **heap** implementation is overall more suitable since it has less order of growth time complexity on the insert operation than an ordered array. (This is just an assumption as we don't have enough details; f.x. initial structure could be fixed which would defy this assumption).*

6. Suppose you want to use 10^3 put operations and 10^6 get operations on a symbol table. Which implementation (that we have considered so far) could you choose and why? More generally, explain how the relative numbers of get and put operations affect the choice of a symbol table implementation.

Solution. *Let's explore the implementations we have considered so far, exploring the average-case time complexity of both the get and put methods:*

operation	***Binary Search Tree***	***Unordered List***	***Ordered Array***
get	$\sim lgN$	$\sim N$	$\sim lgN$
put	$\sim lgN$	$\sim N$	$\sim N$

Since only a single put method is used per 1000 get methods on the given symbol table, the put method becomes practically irrelevant in respect the get method. We therefore first and foremost consider get operation time complexity. According to table above, the minimum time complexity for get operation is achieved by both binary search tree and the ordered array implementations. However, the binary search tree also offers a cheaper put operation which is *then used as a tie breaker in this situation. Hence the* ***Binary Search Tree*** *implementation would be the preferred choice.*

More generally, the constant multiples matter in time complexity, thus deciding on which implementation to use depends greatly on how often a method is used in relation to number of uses of other methods. When method A is used relatively little to method B, method A becomes much less relevant to time complexity and achieving a less time-complexity should be considered first and foremost for method B, then secondly for method A as demonstrated in the put and get methods above.

Let's just look at tilde notation nature to clarify; the tilde notation denotes the most significant time complexity by disregarding the lower order items. Say we have time complexity A for method A and time complexity B for method B. Method A is executed c times and method B is executed k times whereas k is a much larger constant than c. Thus we have overall time complexity of $cA + kB$ for our program. For a tilde notation, the lower order item is disregarded. We want the constant k and it's multiples to be disregarded instead of c and it's multiples. Hence we want B to be the lesser order item so that it would be disregarded along with larger constant k to achieve tilde notation of $\sim cA$. Therefore, it is desirable to choose an implementation that has less order time complexity on method B than on method A. This proves that relative numbers of operations affect symbol table implementation choices.

7. The collection data type OrderedBag supports two operations: add(), which inserts an element into the bag (just like in the Bag data type); and removeSmallest(), which removes the smallest element from the bag and returns it. Explain why no comparison-based implementation of OrderedBag is possible where both operations are require at most $\sim 1/4lg(N)$ comparisons in the worst case.

Solution. *Since we have an ordered implementation, comparisons are necessary for the add function so that we place each element in the correct order into data structure. Also, either comparisons or N-exchanges (in the worst case) of elements are necessary for the removeSmallest function. The lowest possible worst-case time complexity of N comparisons is lgN. It is impossible to have less worst-case time complexity than that when dealing with comparison - thus an ordered implementation such as OrderedBag can not have methods add and removeSmallest that require time complexity at most $\sim 1/4lgN$ comparisons. The time complexity in the worst-case will always succeed this value.*

8. Give and implement a non-recursive version of the **put()** operation on BSTs.

Solution. *Following is an untested java code for an iterative BST put operation based on recursive implementation from the book and it's methods (such as compareTo( ) method). The implementation does not accept multiple keys and updates corresponding node when trying to insert duplicate keys:*

```
// Based on recursive version of a Binary Search Tree
// And the implementations virtual methods
// page 399 in the book
public void BST_put (Key key, Value value) {

        // The empty tree is a special case
        // then new node needs to be root
        // No further actions necessary so we return
        if( root == null) {
            root = new Node( key, value ); return;
        }

        // Create current node for iterations
        // Also need to preserve parent node for later insertion
        Node curr_node = root;
        Node parent_node = root;

        // Parse parent node of new node iteratively
        // To know where to insert new node
        while( curr_node != null ) {

                // First parent_node needs to be curr_node
                // before we assign curr_node in current iteration
                parent_node = current_node;

                int cmp = key.compareTo(curr_node.key);

                // If key is larger we search right subtree
                if ( cmp > 0 ){ curr_node = curr_node->right; }

                // If key is larger we search left subtree
                else if ( cmp < 0 ){ curr_node = curr_node->left; }

                // If compare values are equal
                // curr_node value is updated
                // No further actions necessary so we return
                else { curr_node.value = value; return; }
        }

        // Insertion after finding parent node:
        // Check whether node is left or right child
        // Then insert new node at the right link
        int cmp = key.compareTo(parent.key);
        if ( cmp > 0 )
            parent_node->right = new Node( key, value );
        else if ( cmp < 0 )
            parent_node->left = new Node( key, value );
}
```