

1. This question asks you big O questions about binary search trees and red-black trees.  $n$  is the number of nodes in the structure. “Worst case” refers to a case where the tree has developed in such a way that the `insert`, `search` and `delete` methods are as inefficient as possible, given the structure. Please insert the correct one of  $O(1)$ ,  $O(\log(n))$ , or  $O(n)$  in each of the slots. As you enter your answers, please omit the  $O()$ . For example, if your answer is  $O(\log(n))$ , just write down  $\log(n)$ .

- (a) What is the height of the tree?

	expected case	worst case	
binary search tree			[4]
red-black tree			

- (b) How many leaves does the tree have?

	expected case	worst case	
binary search tree			[4]
red-black tree			

- (c) A node that has at least one non-empty child is called an “internal node”. How many internal nodes does the tree have?

	expected case	worst case	
binary search tree			[4]
red-black tree			

- (d) A node that has two non-empty children is called a “full node”. How many full nodes does the tree have?

	expected case	worst case	
binary search tree			[4]
red-black tree			

2. This question asks you big O questions about unordered linked lists, ordered linked lists (in both increasing and decreasing order), unordered arrays, ordered arrays (in both increasing and decreasing order), hash tables, binary search trees, red-black trees and heaps.  $n$  is the number of data items in the structure. The data is of type  $T$  implements `Comparable<T>` and you may assume that a comparison of two items takes time  $O(1)$  and that an assignment to a variable of type  $T$  takes time  $O(1)$ . “Worst case” refers to a case where the structure has developed in such a way that the `insert`, `search` and `delete` methods are as inefficient as possible, given the structure. You may assume that for linked lists we only have one reference, and that to the first node in the list. You may assume that the size of the array for the hash table is such that it is approximately half full and that hash clashes are resolved by linear rehashing. You may assume that the heap is a **max-heap**. The defining property of each data structure must be maintained (so that, for example, after a deletion from a heap, the resulting structure must still be a heap).

Please choose the correct one of  $O(1)$ ,  $O(\log(n))$ ,  $O(n)$ ,  $O(n \log(n))$ ,  $O(n^2)$ ,  $O(2^n)$ , or whatever for each of the slots. You are giving a bound for the number of **operations** required, where an operation can be any of the basic machine operations: a comparison, an assignment, or a dereferencing, etc. As you enter your answers, please omit the  $O()$ . For example, if your answer is  $O(\log(n))$ , just write down  $\log(n)$ . In each case I have provided the answers for the decreasing ordered linked list.

- (a) How many operations are needed to find the largest element?

	expected case	worst case
unordered linked list		
increasing ordered linked list	$n$	$n$
decreasing ordered linked list		
unordered array		
increasing ordered array		
decreasing ordered array		
binary search tree		
red-black tree		
hash table		
heap		

[9]

- (b) How many operations are needed to delete the largest element?

	expected case	worst case
unordered linked list		
increasing ordered linked list	$n$	$n$
decreasing ordered linked list		
unordered array		
increasing ordered array		
decreasing ordered array		
binary search tree		
red-black tree		
hash table		
heap		

[9]

(c) How many operations are needed to delete the smallest element?

	expected case	worst case
unordered linked list		
increasing ordered linked list	1	1
decreasing ordered linked list		
unordered array		
increasing ordered array		
decreasing ordered array		
binary search tree		
red-black tree		
hash table		
heap		

[9]

(d) How many operations are needed to delete the top ten largest elements?

	expected case	worst case
unordered linked list		
increasing ordered linked list	$n$	$n$
decreasing ordered linked list		
unordered array		
increasing ordered array		
decreasing ordered array		
binary search tree		
red-black tree		
hash table		
heap		

[9]

(e) How many operations are needed to delete the top  $\frac{n}{10}$  largest elements?

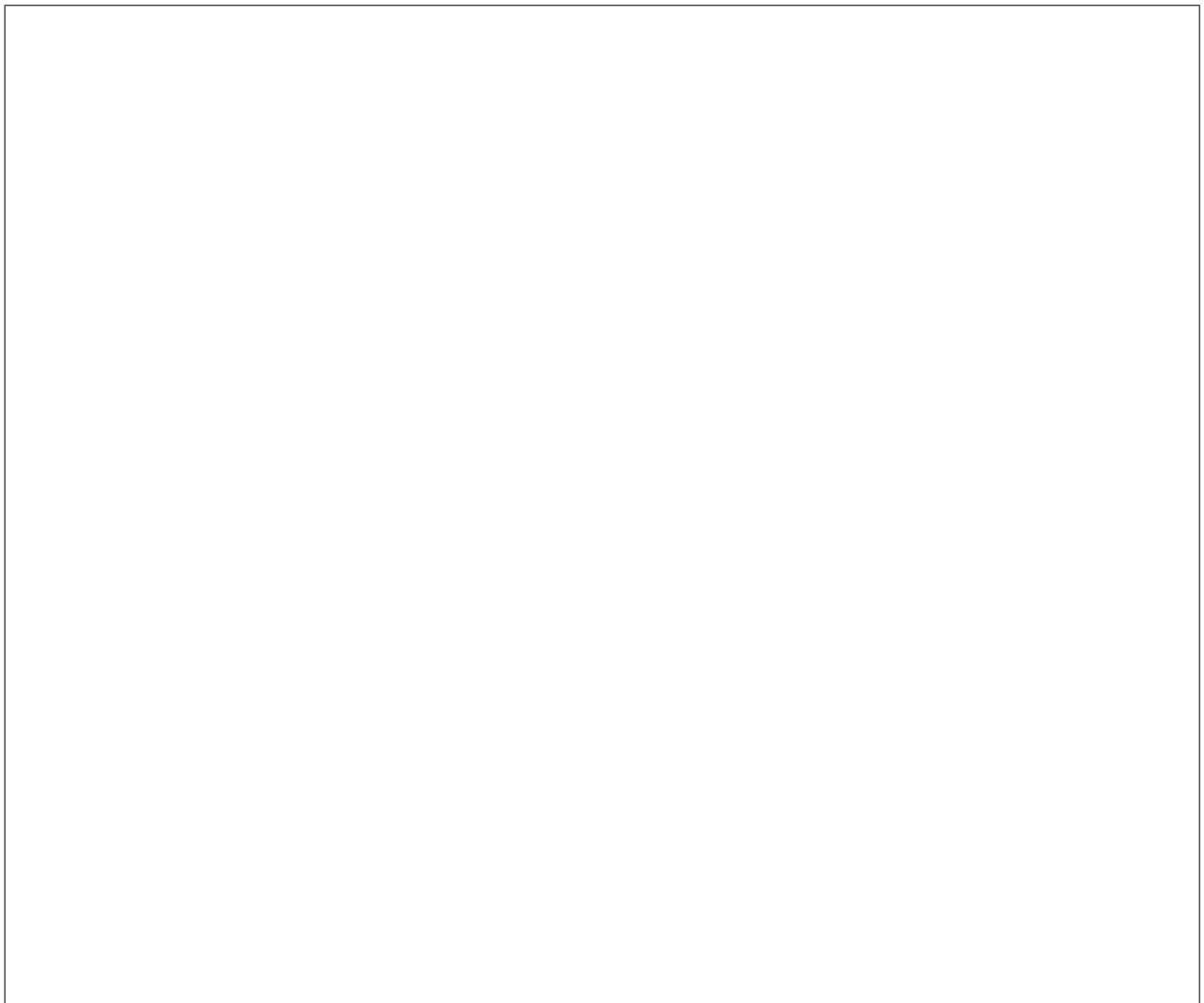
	expected case	worst case
unordered linked list		
increasing ordered linked list	$n$	$n$
decreasing ordered linked list		
unordered array		
increasing ordered array		
decreasing ordered array		
binary search tree		
red-black tree		
hash table		
heap		

[9]

3. I am building a Binary Search Tree of **Strings** with **int** counts (each node stores a string, as the sorting key, and an integer count of how often the string has been encountered), and I am just using straightforward insertion (no fancy balancing as in AVL trees or red-black trees).

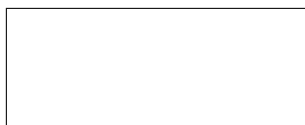
(a) Draw the tree resulting from the successive insertion in the given order of the following elements into an initially empty binary search tree:

is let i me have my handle not  
see hand dagger come still the thee  
this clutch thee before see i and a  
me thee toward i which yet



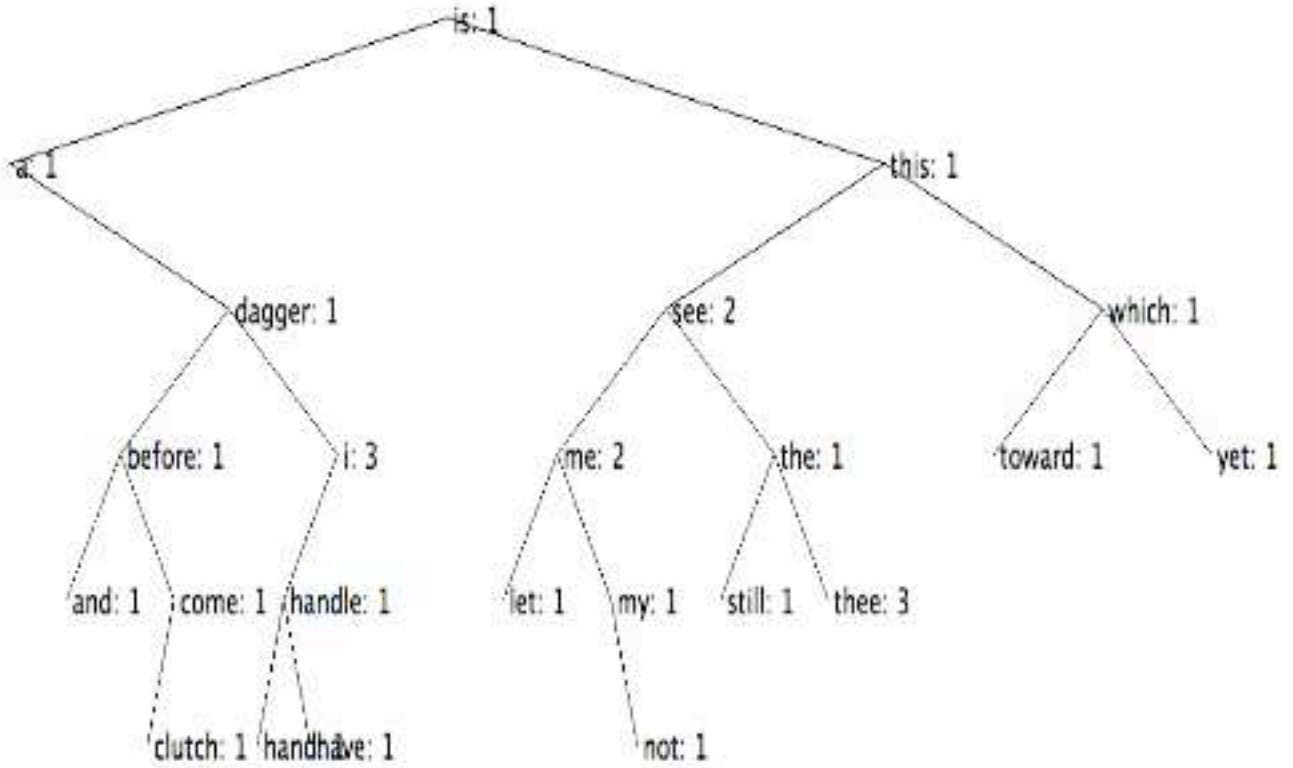
[8]

Would you describe this as a best, an average, or a worst case tree for this collection of words?



[2]

- (b) If the same words had been inserted but in a different (**hint:** more literary) order, the binary search tree might have been as shown below. (Notice that in my rather poor quality image “hand: 1” and “have: 1” are overlapping, and the “a: 1” immediately below and to the left of “is: 1” is hard to read):



Give an order in which the first nine words might have been inserted in order to give rise to this tree.

[8]

Would you describe this as a best, an average, or a worst case tree for this collection of words?

[2]

4. Here are two methods in the class Question4:

```

public boolean checkWithStack(Vector<String> candidate) {
    Stack<String> myWords = new Stack<String>();
    int i = 0;
    while (i < candidate.size()/2) {
        myWords.push(candidate.elementAt(i));
        i++;
    }
    if (candidate.size() % 2 != 0) // if odd number of words
        i++; // ignore the middle word
    while (myWords.pop().equals(candidate.elementAt(i))) {
        if (i == candidate.size() - 1) return true;
        i++;
    }
    return false;
}

public boolean check(String s) {
    Scanner sc = new Scanner(s);
    Vector<String> words = new Vector<String>();
    while (sc.hasNext()) words.add(sc.next());
    return checkWithStack(words);
}

```

(a) Do the following calls to `check(String)` return true or false:

the call	true or false or <b>error</b>	[points]
<code>check("hello hello")</code>		[2]
<code>check("I am what I am")</code>		[2]
<code>check("hello hello hello")</code>		[2]
<code>check("check")</code>		[2]
<code>check("a man a plan a canal panama")</code>		[2]

(b) Now replace the `Stack` with a FIFO structure, such as a `LinkedList` that implements `Queue`. Also replace `push` by `offer` and `pop()` by `poll()`. (Recall that in Java, we use the term **offer** for **enqueue** and **poll** for **dequeue**.) Do the following calls to `check(String)` return true or false:

the call	true or false or <b>error</b>	[points]
<code>check("hello hello")</code>		[2]
<code>check("I am what I am")</code>		[2]
<code>check("hello hello hello")</code>		[2]
<code>check("check")</code>		[2]
<code>check("a man a plan a canal panama")</code>		[2]