**xQ Please fix edit permissions - NO :P**

**COMP3506/7205: 2020 exam answers**

[**UQAttic**](https://uqattic.net)

[**Other exam papers**](https://drive.google.com/drive/folders/1bK6I5-OOEXJz7qLbGKwSVoxWqqnmewuU)

Please **contribute** to these documents.

If you're looking for an effective way to familiarise yourself with the course material, you can't go past collaborating with fellow students. We have laboured to put these up, and so at the very least point out where you think we are wrong!

You'll get more out of the course, you'll do better in the exam, and other students will benefit from your input as well.

**To get editing permissions**, visit [UQAttic.net](http://uqattic.net) and provide us with your Google Account address by clicking on the “Get editing permissions” button on the front page.

**Style.**

Type answers in blue beneath each question.

If you're unsure of your answer, highlight your answer text then hit Ctrl+Alt+M to create a comment beside the text. Once you're satisfied with the answer, click the "Resolve" button on the comment.

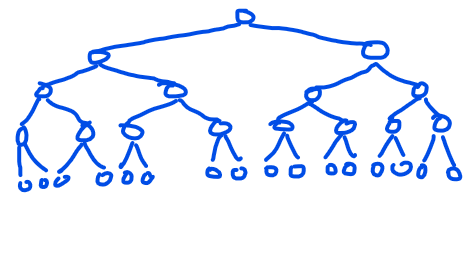
If you want some extra explanation from someone else on their answer, highlight the other person's answer and repeat the procedure above.

-100 Bruh who made the whole thing green

# **Question 1 (2 marks)**

What is the maximum possible number of external nodes in a binary tree with height 4? Give a brief description by drawing a resulting tree.

h

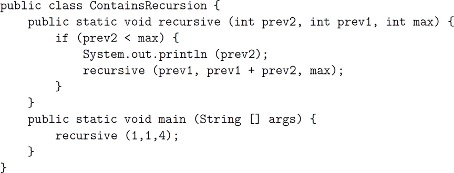


The height of the binary tree is the height of the root node - the maximum number of edges from a leaf node to the root node. This means that a binary tree of height 4 will have a maximum of 2^4=16 external nodes . +6

# 

# **Question 2 (1 mark)**

Consider the following Java program, containing a recursive method. When the class ContainsRecursion is run, the final line of output will be**:**



First Recursive Call -> recursive(1, 1, 4):

Prev2 = 1, Prev1 = 1, Max = 4

(prev2 < max) == (1 < 4) -> **print(1)**

call recursive(1, 2, 4)

Second Recursive Call -> recursive(1, 2, 4)

Prev2 = 1, Prev1 = 2, Max = 4

(prev2 < max) == (1 < 4) -> print(1)

Call recursive(2, 3, 4)

Third Recursive Call -> recursive(2, 3, 4)

Prev2 = 2, Prev1 = 3, Max = 4

(prev2 < max) == (2 < 4) -> print(2)

Call recursive(3, 5, 4)

Fourth recursive call -> recursive(3, 5, 4)

Prev2 = 3, prev1 = 5, max=4

(prev2 < max) == (3 < 4) -> print(3)

Call recursive(5, 8, 4)

Fifth recursive call -> recursive(5, 8, 4)

Prev2 = 5, Prev1 = 8, max = 4

TERMINATE RECURSION

The final line of the output is `3` -> You can confirm this using the online compiler here <http://tpcg.io/3Q9397> +4

**Question 3 (4 marks)**

1. Given two algorithms, **A** with O(**n**) time complexity, and **B** with Θ(**n** log**n**) time complexity, testing shows that algorithm **B** is faster than algorithm **A** in practice for all datasets tested. Give two possible reasons why this may be. (2 marks)

Algorithm A -> O(n) Upper Bound

Algorithm B -> Θ(n log n) Tight Bound (Upper Bound and Lower Bound)

Reason 1 - Asymptotic Analysis

* Asymptotic analysis (i.e. Big-O, Big-Omega and Big-Theta) describes the performance of algorithms and data structures as n-> infinity.
* The datasets used for testing may not have a sufficiently large size (i.e. value of n) to exhibit the true nature of the algorithms’.

Reason 2 - Lower Order Terms

* The lower order terms in the functions may be dominating the time complexity of these algorithms’ at smaller values of n.

+2

1. Hidden constant value like (10^100)\*n compare to n\*logn

‘ 2.input size might two small>

1. Given two algorithms for performing a task, is it ever reasonable to use the algorithm with worse asymptotic performance? Why or why not? (2 marks)

Yes, it is valid to use algorithms’ with worse asymptotic performance - the asymptotic performance of an algorithm describes the performance of algorithms with very large input sizes.

It is very much possible that an algorithm with a worse asymptotic performance performs better with smaller input sizes - it depends on how large the typical inputs to the algorithm are in the specific use case.

An algorithm with a better asymptotic performance may be compensating by using more memory, thus making it undesirable.

Could work better on certain data types/instances, e.g. use adaptive sorting algorithm if the data is mostly sorted

+1

# **Question 4 (2 marks)**

R1(n) = 4 R1(n/2) + O(1)

R2(n) = 2 R2(n/4) + O(1)

Explain which one is 2 and why? (2 marks)

】

R1(n) = 4 R1(n/2) + O(1)

= 4 (R1(n/4) + O(1)) + O(1)

= 4R1(n/4) + 4O(1) + O(1)

= 4R1(n/4) + 5O(1)

R2(n) = 2 R2(n/4) + O(1)

We observe here that the coefficient in front of R2 is smaller than that in front of R1. Additionally, the size of the input passed to the next iteration of the recursive function is halving in R1 and ??quartering?? In R2 -> this means that R2 will decrease in size significantly faster than R1 (??exponentially??)

R1(n) = 4R1(n/2) + 1 = 4(R1(n/4) + 1) + 1 = 4((4R1(n/8)+1) + 1) + 1 = …

= 1 + 4 + 16 + … (totally log2(n) elements) + 4^k = (1 - 4^(log2(n))) / (1 - 4) = n^2 / 3 - ⅓ = O(n^2), because it’s a geometric series

and 4^(log2(n)) = 2^2^log2(n) = 2^log2(n)^2 = n^2

R2(n) = 2R2(n/4) + O(1) = 2(2R2(n/16) + 1) + O(1) = 2(2(2R2(n/32) + 1) + 1) + O(1) …

= 1 + 2 + 4 + 8 + … (totally log4(n) elements) + 2^k = (1 - 2^(log4(n))) / (1-2) = (1/2)\*n^(1/2) - ½ = O(n^1/2), because it’s a geometric series and 2^(log4(n)) = 2^1/2^log2(n) = 2^log2(n)^1/2 = n^1/2

If n is large, O(n^1/2) < O(n^2), so R2 is faster.

+2

7

Since O(n^½) < O(n^2), R2(n) is faster 👆

+1+1

# **Question 5 (2 marks)**

Draw a table to represent the “last occurrence” function for the pattern P="

baccarat", which is used as part of the Boyer-Moore pattern matching algorithm.

B a c c a r a t

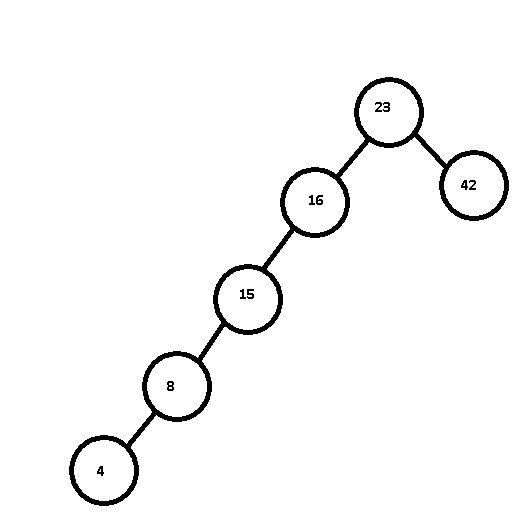
0 1 2 3 4 5 6 7

| Character | b | a | c | r | t |
| --- | --- | --- | --- | --- | --- |
| L(c) | 0 | 6 | 3 | 5 | 7 |

+4

# **Question 6 (4 marks)**

A splay tree is populated by inserting into an empty tree the keys: 4, 15, 8, 16, 42, and 23, in that order.

1. Draw the final tree. No need to include the external leaves in the drawing. (2 marks)

-2 +1 +1 +1 +1+1

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

1. What is the order that nodes are visited during a post-order traversal of the resulting tree? (1 mark)

4, 8, 15, 16, 42, 23 +1

This is not even a traversal? Post order of above would be 23, 42, 16,15,8,4（no way, post order must make the root last）-1 -1

+1 +1 +1+1

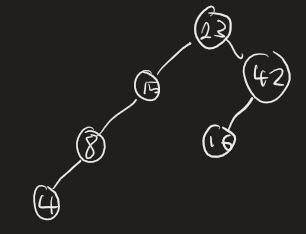
1. Draw a different splay tree with the same post order traversal. (1 mark)

Is there an answer?

I think it is impossible: +3

(Confirmed by Max, it’s impossible)

Why?



The above is a BST… U can’t just set 23 as root and order the rest nodes, the question asks specifically for a splay tree. U need to show work out, can’t just leave a BST here. The question now becomes part (a), with the ordering changing from 4, 15, 8, 16, 42, and 23 to 4, 8, 15, 16, 42, 23. And the result MUST be the same. So it is impossible to draw a second splay tree.

3

This should work, I read the question as it doesn’t matter how the tree comes about it just wants a tree with the same post order traversal? 16<23 Yeah true? So this doesn’t work,right child of 23 must larger than 23 42>23 16 is also belong to right child of 23

`

# 

# **Question 7 (the 2 marks)**

We defined a hash function h(s) = s.size(), for a hash table which stores a set of strings.

1. Explain why h(s) is not a good hash function. Note: size() returns the length of the string. (1 mark)

The hash function h(s) = s.size() is not a good hash function for a hash table - a hash function. When designing a hash table, we should try to develop a hash function that minimises the number of s (i.e. distinct inputs with the same output).

This means that the strings “cat” and “dog” have the same hash value under this particular hash function even though they’re evidently different.

9u

+1

1. Suggest an alternative hash function or technique that would perform more suitably (and why?). (1 mark)

We could use polynomials or primes to construct a more appropriate hash function. For this example, let ASCII(c) return the ASCII value of the character. Additionally, let ASCII(a) = 65, …

We could define our hash function as follows:

h(s) = sum(ith prime number \* ASCII(s[i]))

This hash function takes t me number multiplied by the ASCII value of the ith character in the string s. This ensures that strings with the same length, comprised of the same characters in a different order have a different hash value.+1

Referencing 2021 Project B, let’s suppose we have a well defined hashcode function h(s). Where:

* h(s) = 1 + (1 + c1) + (1 + c1 + c2) +.....+ (1 + c1 + c2 +....cn)
* saasWhere c is the ascii value of that character.

This ensures that if given two strings of the same size “dog” and “god” we eliminate the chances of collisions twofold, as the size is not a prime factor in determining h(s) and the ordering of the letters is important.

Suppose “dog” (with c1 = d, c2 = o, c3 = g), god(with c3 = g, c2 = o, c1 = d)

h(“dog”) = 4 + 3c1 + 2c2 + c3 and h(“god”) = 4 + 3c3 + 2c2 + c1

Therefore “dog” and “god” even with the same letters and number of letters, even with the same letters used, the order allows for a different hashcode to be generated.

^ This is called the cumulative component sum function

# **Question 8 (4 marks)**

Arrays store a set of values by index using O(*n*) space, where *n* is the largest index in use. In some cases,most indices of the array are *null*, wasting space; such arrays are called **sparse arrays**. In other words, the number of non-default values stored *m*, is much smaller than the largest index *n*.

-

1. Describe how you would efficiently implement a sparse array using a hash table. What would be the worst-case time complexity of standard array operations (insert, delete (moving indices of elements later in the array after deletion), and search an item) in Big-O notation, if we implement sparse arrays using hash tables? (2 marks)

We could use a hash table to construct a dl of a sparse array. An entry’s index in the sparse array would be used as the attribute to be hashed as we want to use the entry index to find / retrieve / update the values in the hash table.

In the worst-case, insertion would take O(n) - this occurs in the case where the hash table has a very high load factor (i.e. large proportion of cells filled), as we have to iterate through every cell to try to find the element.

In the worst-case, deletion would take O(n) time for the same reason as stated above.

In the worst-case, search would take O(n) time for the same reason as stated above.

* Would these not be O(m) because you are only putting in non-default values into the hash table.??? +1 +1 +1 + 1 +1
* O(m + k) if we use open hashing to handle collisions??? -1

O(m) is the worst case no?

Can do this in O(m) time if you set the size of the hashtable to be proportional (e.g. two times) the size of the number of inputs. This reduces the size of the hashtable to O(m). You resize if the load factor is too high. This is probably smarter, since if you use O(n) space you might as well use an array.

1. Describe how you would efficiently implement a sparse array using a linked list. What would be the worst-case time complexity of standard array operations in Big-O notation, if we implement sparse arrays using linked lists? (2 marks)

However, the best-case and worst-case insertion, deletion and search for this data structure would be O(n) as we have to iterate through every entry in the linked list.

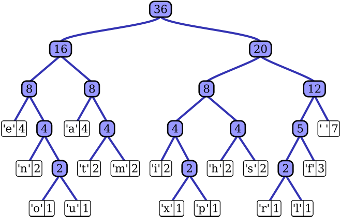
Could insertion be O(1) since we add to tail, we don’t care about order in this situation? **+ 1 +1**

- why best-case? +1

Worst case insertion, delete, search should be O(m) depending on how you implemented it. If you are storing the items in index order, all functions are O(m) because you need to iterate through the whole list. If you don’t care about order, insert could be O(1) but all other functions are still O(m).

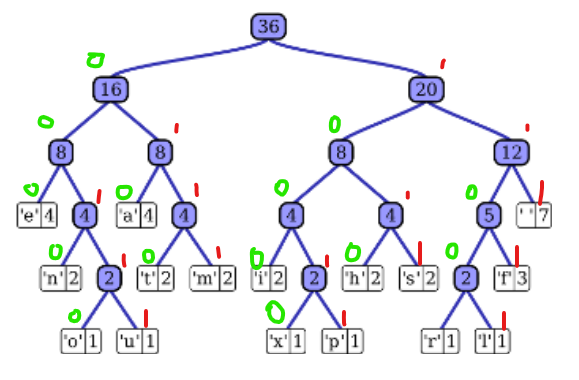
# 

# **Question 9 (2 marks)**Below is a Huffman tree generated from the exact frequencies of the text: "this is an example of a huffman tree"



Using the Huffman tree above, determine the binary code and count the number of bits required to transmit the text “fill pool” (do not forget the space separating the words). Note: Left branches represented a 0 bit and right branches represent a 1 bit.

The code derived from the huffman tree is derived from traversing from the root node to the leaf node containing the desired character

9

F: 1101 I: 1000 L: 11001 “ “: 111 P: 10011 O: 00110

CODE = 1101 1000 11001 11001 111 10011 00110 00110 11001 +1+1+1

F I L L “ “ P O O L

# BITS = 4 + 4 + 5 + 5 + 3 + 5 + 5 + 5 + 5

= 41 bits +1

Why not add up all the 1 bits? Since it’s the total number of bits needed to be transmitted (0 or 1), not just the 1 bits

# **Question 10 (5 marks)**

For each of the following scenarios, suggest an efficient and suitable sorting algorithm (out of the ones we learned in this course), and explain why you chose it. Note: n potentially is very large.

1. We have an array consisting of n objects in random order. The objects are comparable, and not necess-arily numbers. (1 mark)

Assuming that we are able to modify the input array, we could use In-Place Quick-Sort to sort the array. This means that we can sort the array with O(n log n) average time with no additional storage space required.

Another reason for choosing Quick-Sort is that it handles sorting large inputs well as compared to other other algorithms.

Shouldn’t it be radix sort? It is the only one that can sort non-numbers by using lexicographic sort.

As long as the items are comparable, quick sort will work on it. For example if they were words, you would just do each comparison lexicographically. How the comparison is done does not seem to matter for the question.

Wouldn’t merge sort be preferable? Quick sort is expected nlogn, merge sort is guaranteed nlogn. +2 +1

Quick-sort is generally O(n log n) but keeps memory usage down which is useful as question says n is very large

Quick-sort is fine. It’s typically faster due it to being better for cache as it is in-place. However, saying **quicksort uses no additional storage space may be interpreted as stating that it is O(1) space which is wrong**. **The recursive calls take up O(n) space.** Saying mergesort should be perfectly fine too. Mergesort has the benefit of being a stable sort which is often desirable.

1. We have an array consisting of n objects. The objects are comparable. Apart from d items (d< log2*n)* that are in random positions in this array, all the other items are in sorted order. (2 marks)

Since the data to be sorted is mostly sorted, we could use an adaptable sorting algorithm (which makes use of the fact that the data is partially sorted). Adaptive sorting algorithms such as Insertion Sort typically run in O(n^2) time, but as a proportion of the data that is sorted approaches 100%, the run-time of the algorithm approaches O(n).

1. We have an array consisting of n random integers in the range of 0 to d, with d< log2*n*. (2 marks)

Since we are sorting integers, we can use a sorting algorithm such as Radix sort - this utilises the fact that the data we are sorting are integers. This is beneficial as the time complexity for radix sort is O(n + d). However, Radix sort requires the use of an external data structure for storing this data.

If space complexity is a constraint, we could always revert to a comparison-based sorting algorithm such as quick-sort.

Should radix time complexity be O(n \* d) instead of O(n + d) ? and it’s space complexity O(n + d)? +2 Yes

Radix sort time complexity is O(n \* d), where d is the num of digits in the binary number. Check Week 4 Tut, Slide 31. Yes, +1

I’m not too sure (+1)

Bucket sort instead of radix sort? +1 why not bucket? If we know items are integers, more efficient to use radix than have n buckets

The answer is using bucket sort instead of radix sort, but they called it radix. Since the number of buckets will be d and there are n elements, the time complexity will be O(n + d).

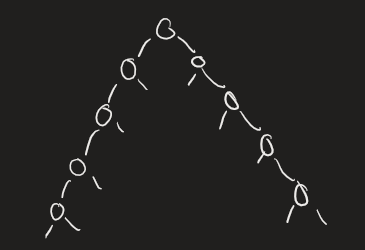
Radix sort is useful because the time complexity is O(b \* n), where b is the maximum number of bits in an integer. For an integer of size d, the number of bits in it will be something like log(d), so the most efficient sorting algorithm would be radix.

Bucket sort has better performance than radix sort if radix sort is based on digit. Radix sort will need log(d) buckets for the digits. The actual bucket sort that is used by radix sort is O(n). So it is O(n\*log(d)) = O(nlog(logn)). Bucket sort runs in O(n+d) which is O(n + log n). V

# **Question 11 (4 marks)**

1. **Given a semi-balanced binary tree with n nodes, where the number of leaves in the corresponding left and right subtrees differs at most by one. Explain the worst-case height of the tree. Note: Write theBig-O with respect to the number of nodes (n). (2 marks)**

**O(log n) - this is an AVL tree.**



The number of leaves in the left and right subtree of this tree is equal, so it satisfies the semi-balanced property. But, the full height is n/2, so the worst case time is still O(n)

Only the first two sub trees of the root are balanced, sub trees on different levels are not semi-balanced. For example, the root of the right tree has a right sub tree with 4 leaves and a left subtree with 1 leaf. I think height is O(log(n)).

No they are balanced (0 and 1 leaf), O(n) is correct.

If it is O(n) which means that it will not be an AVL tree anymore!

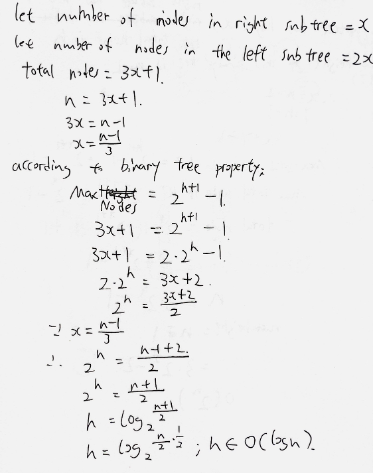
Don’t think the tree you gave here is semi-balanced. The property needs to hold for all nodes in the tree. Let a = the left child of the root, then the number of leaves in a.right = 1, but the number of leaves in a.left is 3. I think this means that height is O(logn)

A semi-baa

1. Given a semi-balanced binary tree with n nodes, where the **number** of nodes in the left subtree is at most twice the number of nodes in the corresponding right subtree. Explain the worst-case height of the tree. Note: Write the Big-O with respect to the number of nodes (n). (2 marks)

O(log n) +1

O(log(n)). Suppose the worst case height is k. Have every node keep the difference left subtree is twice right subtree. Suppose the root has 3^k a children overall, then the left sub has 3^(k-1)\*2 and the right sub has 3^k-1. Continue this inference until the bottom, where each node has 1 node as right sub and 2 as left sub. So overall the height would be k+1, which 3^k = n. So the height is still O(log(n))



My answer is O(logn) as shown above.

I got O(n), since left subtree could have 0 nodes every time. 0 < 2x, where x is number of nodes in right subtree. So you could have a long line just along the right, which is O(n) height. ← This is correct (Ed#808 - scroll to the bottom), but Minhao thinks it wasn’t what the question intended to ask. But definitely still correct.

# **Question 12 (4 marks)**

Given an unweighted graph G with **n** vertices and the adjacency matrix A:

1. Given a positive integer **k**, propose an algorithm (give a pseudo-code or explain in words) to calculate the number of walks (vertices/edges can be revisited) consisting of exactly **k** number of edges between vertices *i* and *j*. Your algorithm should return 0 if there are no walks of length **k**. (3 marks)

This question is present in one of the tutorial sheets in 2021. By computing A^k and indexing into the appropriate cell in the matrix, we can find the number of walks of length k from vertex i to j

Algorithm numWalks(i, j, k, A)

Input: i - starting vertex for walks

j - ending vertex for walks

k - walk length

A - adjacency matrix.

return matrix\_power(A, k)[i][j]

WTF is matix\_power???

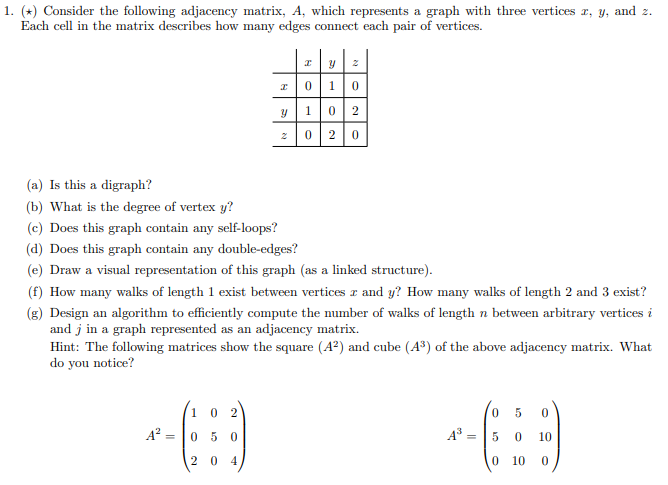
I don’t think this ever came up in a tutorial? Wk 9 tute Q1.g

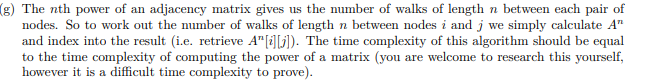
Tutorial question is pase answer beforehand, this question would’ve been so tough +2 z+1

So, someone can include pseudo code? I think it’s based on BFS right?

ted on next page

Bro without knowing th





If you study this tutorial question then it is not so tough.

1. What is the worst-case time complexity of your proposed algorithm in terms of

**n** and **k**? and explain why. (1 mark)

In the tutorial, it is stated that the time complexity of such a method is directly proportional to the amount of time required to compute the matrix power. That is, the time complexity of the algorithm is the time complexity of the matrix\_power function

# **Question 13 (4 marks)**

Work out the most efficient Big-O time complexity of finding pth smallest element in a BST with **m** elements. Hint: The Big-O will be in terms of **m** and **p.** Describe in words your proposed algorithm.

In order traversal? Worst case will be O(m)

.

Most efficient BST configuration is a perfect tree. Given this, a search to each external node is O(logm), if you would like to find the pth element, then p of these searches is required. Therefore, **O(plogm)**

The question says nothing about the tree being a perfect tree so we cannot assume the height is O(logm). When searching, we are looking for a specific value but we do not know what that value is so binary search p times would not make sense. An inorder would find the elements in increasing order.

AAAAAAAAAAAA

- But the question states that the Big-O will be in terms of m and pI

-I would say it is just O(p) ? but hey I could be wrong, because you would in order traverse p times, because this would give you the pth smallest element.

Construct an array with size m, add nodes of tree in in-order traversal order. This takes O(m). Iterate over array p times to find p smallest element = O(p). Overall complexity is O(m + p). I think this is the solution the question is looking for, alternatively you do an in order traversal, and return the p-th node found. This is still O(m + p) since the worst case for height is O(m) (e.g. all nodes in left children), but it has a better best case, as you don’t always have to iterate over the whole graph. Pseudo-code for the 2nd algorithm would look something like this:

Algorithm inOrder(G, p)

Initialise i = 0

Return inOrder(G, G.root, p)

// in order is (left, root, right)

Algorithm inOrder(G, n, p)

If n is not Null then

inOrder(G, n.left, p) // recurse into left child first (in-order)

// examine root

If i == p then // found our ‘pth’ node

Return n.value

else

i++

inOrder(G, n.right, p)

But if you construct an array that represents the in-order traversal, cant you just go A[p] which takes O(1) time, so O(m) time overall?

It should just be O(p). We can execute an in-order traversal of the graph, where each time we would append a node into the array, we instead decrement a counter we initialise to be p. Once this counter is zero, we know we’ll have the pth smallest element without needing to reference an array, nor even know the size of the tree itself.

But to perform an inorder traversal we would need to traverse all the way down to the left subtree to get the leftmost node and then only can you decrement the p counter. Which means that it could still be O(p + m) ??

Note that O(p+m) is O(m) as p < m. The time complexity is O(min(depth of minimum element (leftmost element), p)). This is as even if P = 1, we have to go down the binary search tree to find the depth of the minimum element. This depth in the worst case is m as a tree could be a left-skewed linked list. Therefore it is O(m), if we had to do it in terms of only m and p.