



**SOFTWARE ASSESSMENT MATERIAL RELEASE**

THEORY QUESTIONS

|  |  |
| --- | --- |
| **SECTION** | **MARK** |
| **1. Theory Questions** | 26 |
| **2. Coding Questions** | 24 |
| **3. Theory Challenge** | 25 |
| **4. Coding Challenge** | 25 |
| **TOTAL** | **100** |

**Important notes:**

* This document shares the ﬁrst section of the Software Assessment which is composed of 6 Software Theory Questions
* It is worth just over a quarter of your assessment mark
* You have 24 hours before the assessment to prepare.
* If any plagiarism is found in how you choose to answer a question you will receive a 0 and the instance will be recorded. Consequences will occur if this is a repeated offence. You can remind yourself of the plagiarism policy [h ere](https://drive.google.com/file/d/1k9UaGOR7hx54QRZ8jvp2jtC4P-8_Rs4F/view?usp=sharing).
* You are **not** allowed to use any online images to support your answers.

**Section 1: Theory Questions [26 points]**

**1 point**

* 1. **The deque module is part of which python library?**

**2 points**

**1.2 What are 2 differences that distinguish a tree from a graph?**

**2 points**

**1.3 Give the deﬁnitions of time complexity and space complexity**

**5 points**

**1.4 Describe the bubble sort algorithm and its complexity. What is guaranteed at the end of the ﬁrst pass?**

**8 points**

**1.5 Explain what LIFO and FIFO are and how each works in practice with a named data structure**

**8 points**

**1.6 What is a Balanced Binary Tree and what would be the best root? Walkthrough how you search using this structure.**

* 1. The **Deque** is part of **collections library**.
  2. 2 differences that distinguish a tree from a graph:

**Different traversal techniques**: trees have three traversal techniques (pre-order, in-order and post-order), graphs have two (breadth-first search and depth first search).

**Presence of a root node:** in a graph we don’t recognize presence of a concept of root, whereas in a tree, there is a unique node, known as a root.

* 1. Under **time complexity** we understand a crucial component of performance analysis for algorithm which calculates the time required for it to run depending on the size of input. For example, in recent homework we had a program that rearranges the cue depending on the command attached to the name of the participant, in this case the length of time the algorithm takes to run would directly depend on the size of the input file, the more we need to rearrange the longer it takes, and time complexity for such function would be 0(n) where (n) is the number of lines in input file.

Under **space complexity** we would understand the total space the algorithm takes depending on the input size, analyzing space complexity requires looking at whether algorithm requires us to create additional data structures or variables. Normally we use o(1) space complexity for cases where the amount of space used doesn’t depend on the input size and the algorithm doesn’t involve creating new variables or data structures.

* 1. **Bubble sort algorithm** works by running through the list over and over again, comparing adjacent elements, swapping them when their order is incorrect. The process stops when the full list is sorted. After each pass, the largest element in the portion of the list that has been unsorted so far moves to its correct position, so **after first pass**, the largest element of our list will be in its correct position, at the very end of the list.

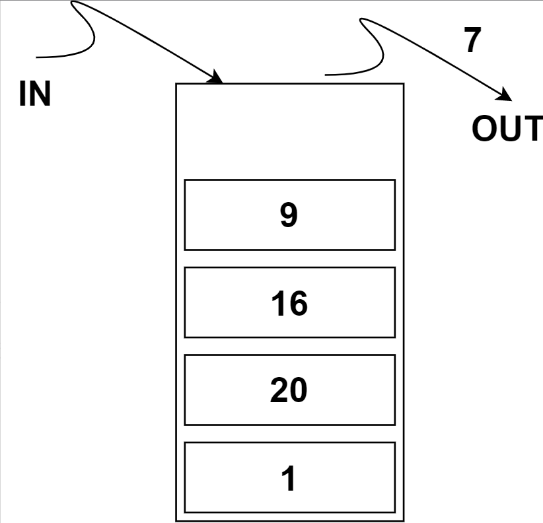
**Space complexity of bubble sort** is 0(1) as it requires fixed amount of space and doesn’t create new variables or data structures.

**The best case of time complexity** for bubble sort is **0(n)** where time it takes to perform will be directly correlated to n, the number of items being sorted, this is usually the case for lists where items are presorted to some extent. And **worst case and average complexity would be 0(n2)**, again n is the number of items being sorted, but it means it takes much longer to sort the lists that aren’t presorted to some extent or reverse the order completely.

* 1. **Explain what LIFO and FIFO are and how each works in practice with a named data structure:**

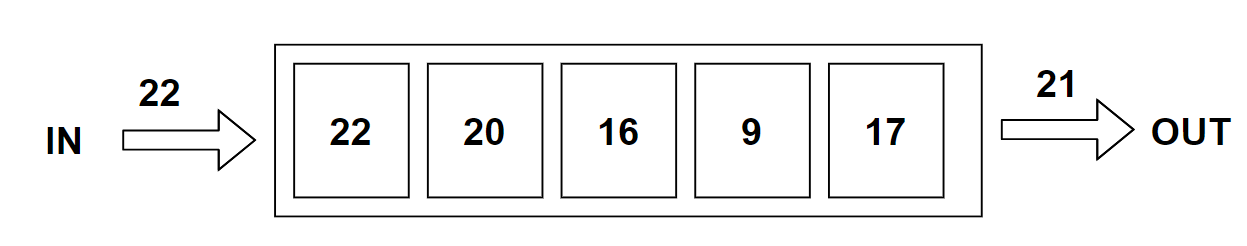
**LIFO – last in first out** principle means last item added to collection is the first one to be removed. Analogy would be a stack of heavy boxes of printer paper, where the typical order would be to remove and open the topmost box, as it is convenient and practical approach.

**The stack data structure** uses LIFO principle for maintaining data, it means if we push the element into data structure and pop it, we would use the last element added, as the point of entry and exit of data is the same.

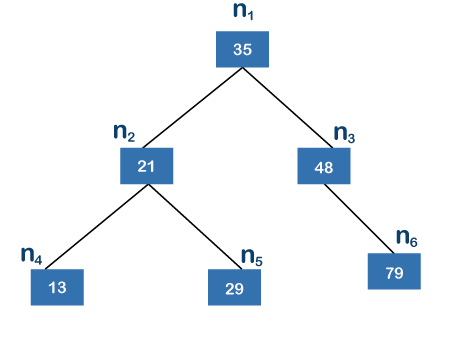


**FIFO -first in first out** principle means we first item added to collection is the first one to be removed. Practical life analogy would be a line to enter the cinema where visitors enter in order, they arrived to the ticket control station.

**Queue data structure** uses FIFO principle for maintaining the data, as elements are added and removed from the queue their points of entry and exit wouldn’t be the same, so the first item added to the cue would be the first one out.



* 1. **Balanced Binary Tree** is also referred to as height balanced binary tree is a search tree that maintains order between the nodes and controls its height. Which means after addition or deletion a balance tree must rearrange itself, maintaining the height difference between left and right subtrees to be no greater than 1, where the height of a tree is the number of edges on the longest path between the root node and the leaf node. Balanced binary trees are commonly used for indexing databases, they allow for efficient search and retrieval of data based on indexed key.



The best root to choose is the element that would allow to maximize the balance on the left and right subtrees, if at all possible, the root should divide remaining elements in two equal-sized subtrees, maintaining optimally balanced structure.

Search process:

Starts at the root node

* If the target element is equal to the current value, search is complete
* If the target element is bigger than current value, search moves to the right subtree
* If target element is lower than the current value, search moves to the left subtree

Search sequence is then repeated going down the tree nodes until the target element is found and search is complete. The search efficiency is achieved by avoiding going through all elements of the tree.