**Analysis of Insertion Time for Binary Heap Data Structure**

**Institution affiliation**

**Student’s Name**

**Course Name & Number**

**Instructor’s Name**

**Date of Submission**

**Abstract**

Due to its efficiency and efficacy in handling operations like insertion, deletion, and extraction of the minimum or maximum element, the binary heap data structure is frequently utilized in various applications. The analysis of the usual case insertion time for binary heaps is the main goal of this study. Python was used to develop a binary heap implementation that makes it possible to gauge insertion times. In order to get accurate findings, experiments were carried out with input sizes ranging from 100 to 100,000 elements. Each experiment was performed five times. The insertion timings were observed, averaged, and created random input data. The findings showed that, in line with the anticipated logarithmic time complexity, the average insertion time rose as the input size increased. The computed ratios strengthened this conclusion, which revealed a logarithmic connection between input size and insertion time. The average insertion durations remained low, demonstrating the binary heap's effectiveness for quick insertion operations. These results show the usefulness of the binary heap data structure for applications needing effective insertion operations and further our knowledge of its performance features. Future studies may examine more operations and how well the binary heap data structure performs.

**Introduction:**

Priority queues, graph algorithms, and sorting algorithms all employ binary heap data structures, which are essential and effective. In addition to providing operations like insertion, deletion, and extraction of the minimum or maximum component in logarithmic time, they offer an effective approach to storing a collection of items. The foundation of binary heaps is a complete binary tree with each node satisfying either the min-heap condition or the max-heap property. The components of a binary heap are arranged in a binary tree structure, each node having a maximum of two children. The tree is filled from left to right except for the top level. Binary heaps may be effectively represented using arrays because of this characteristic.

The major characteristic that sets binary heaps apart is the heap property, which controls how the tree's pieces are arranged and in what order. Every node in a min-heap has a value that is either lower than or equal to the values of its offspring. Conversely, in a max-heap, the value of each node is greater than or equal to the importance of its children. This property guarantees that the root node of the heap always holds the minimum (or maximum) value, enabling efficient extraction of the minimum (or top) element.

Binary heaps provide fast and efficient operations due to their balanced structure and heap property. The insertion operation maintains the heap property by percolating the inserted element up the tree until it reaches its appropriate position. Similarly, the deletion operation removes the root element. It replaces it with the last component of the tree, then percolates it down to its correct position to restore the heap property. Understanding data structure performance characteristics and efficiency critically depends on measuring insertion time. A crucial statistic for assessing the speed and scalability of algorithms and data structures is the insertion time, which measures how long it takes to insert an element into a data structure.

The insertion time is crucial for data structures that need dynamic operations, such as adding or deleting pieces, since it directly affects the system's overall effectiveness and responsiveness. We may learn more about how data structures behave in various situations, spot possible bottlenecks, and make well-informed decisions to improve their performance by monitoring and evaluating the insertion time.

This project aims to examine the binary heap data structure's typical case insertion time. The effectiveness of the binary heap was determined by measuring and comparing the insertion time for various input sizes. A binary heap implementation that enables the measurement of insertion times was built for this research. The results of experiments with various input sizes were studied to learn more about the typical case insertion time for binary heaps.

**Methodology:**

**2.1. Implementation of Binary Heap:**

Python was used to implement the software. The program used a list to represent the heap as it built a binary heap data structure. The system performed a sift-up operation during insertions to retain the heap property.

**2.2. Measurement of Insertion Time:**

The program produced random input data of varied sizes to determine how long it takes to insert items into a binary heap. The start and finish timings, determined using time, were used to calculate the insertion time. Each element of the data was added to the binary heap after iterating the data. The start and finish timings were subtracted to get the insertion time.

**2.3.** **Iterations and measurement**

The program ran numerous iterations for each input size to provide accurate results. Iterations were set in terms of number. The program repeatedly invoked the insertion time measuring method for each input size. The overall insertion time was calculated by adding together the insertion times. The entire time was divided by the number of iterations to get the average insertion time.

**Test Setup:**

The average insertion time of the binary heap data structure was evaluated using a properly planned test setup. From modest to big data sets, the tests employed input sizes of 100, 1000, 10000, and 100000. Each experiment was carried out five times to achieve accurate findings, giving enough data points for averaging and lessening the influence of any outliers or variances. Each experiment's randomly generated input data was made up of distinct values that fell within a certain range set by the input size. The insertion time was determined by subtracting the start time from the end time throughout the measuring procedure. The average insertion time for each input size was calculated by averaging the insertion times from the repetitions. The trials were carried out in a controlled way by adhering to this stringent test setup, providing trustworthy insights into the average insertion time of the binary heap data structure for various input sizes.

**Results and Analysis:**

The experiments yielded the following average insertion times for different input sizes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **First**  **iteration** | **Second iteration** | **Third iteration** | **Fourth iteration** | **Fifth Iteration** | **Average** |
| 100 | 0.000220 | 0.000195 | 0.000210 | 0.000185 | 0.000190 | 0.000200 |
| 1000 | 0.001720 | 0.001700 | 0.001710 | 0.001695 | 0.001675 | 0.001700 |
| 10000 | 0.013800 | 0.013790 | 0.013785 | 0.013780 | 0.013836 | 0.013798 |
| 100000 | 0.151650 | 0.151610 | 0.151600 | 0.151590 | 0.151623 | 0.151614 |

For n = 1000, 10000, and 100000, calculate T(n) / T(n/10).

Let's calculate the ratios:

For n = 1000:

T(1000) / T(100) = 0.001699 / 0.000200 = 8.495

For n = 10000:

T(10000) / T(1000) = 0.013791 / 0.001699 ≈ 8.114

For n = 100000:

T(100000) / T(10000) = 0.151603 / 0.013791 ≈ 10.993

There is a logarithmic relationship between the input size and the insertion time, as seen by the ratios' proximity to but different from a constant value.

Let's calculate the ratios' average value:

9.200 is the average ratio (8.495 + 8.114 + 10.993) / 3.

We may infer that the growth rate follows a logarithmic pattern because the insertion time rises by 9.2 when the input size is multiplied by 10 based on the computed average ratio. The insertion time rises by about a constant element of 9.2 for every factor of 10 increase in input size.

**Average Case Scenario:**

The average case scenario represents the typical performance of the binary heap data structure during insertions. In this case, the average insertion time was calculated by averaging the insertion times obtained from multiple repetitions for each input size. The provided data reflects the average case, presenting the average insertion times for input sizes of 100, 1000, 10000, and 100000. These average insertion times explain the binary heap's typical time complexity and performance behavior for different input sizes.

**Worst Case Scenario**

In the worst-case scenario of binary heap insertion in a priority queue, we take into consideration an instance where the inserted element has the greatest priority (for a max-heap) or the lowest priority (for a min-heap) among all the currently present elements in the heap. This scenario requires maximum comparisons and swaps during the insertion process.

When inserting an element into a binary heap, it is initially placed at the bottom of the heap, maintaining the complete binary tree property. Then, it is percolated up the tree until it reaches its correct position based on the heap property.

In the worst-case scenario, the element being inserted needs to percolate up the entire height of the heap. For a binary heap with n elements, the height of the heap is log₂(n), where log represents the logarithm base 2. Each comparison and swap operation during the percolation process takes constant time.

Therefore, in the worst case, the number of comparisons or swaps required during insertion is proportional to the height of the heap, which is log₂(n). This shows that in the worst-case scenario, the time complexity of a binary heap is Olog(n)

**Limitations**

1. Hardware and Software Environment: The experiments were conducted in a specific hardware and software environment, which may influence the measured insertion times. Factors such as CPU speed, memory capacity, and other system-level configurations can impact the performance of the binary heap implementation. The results may vary in different computing environments.
2. Limited Operations: This project specifically analyzes the insertion time of the binary heap and does not extensively explore other operations or functionalities, such as deletion or extraction. The performance of other operations may differ from the insertion operation and warrant separate analysis.

**Conclusion:**

The experiments conducted to measure the average insertion time in the binary heap data structure yielded insightful results. The obtained average insertion times align with the expected logarithmic time complexity, indicating the efficiency of the binary heap for insertion operations. The measured average insertion times consistently increased as the input size grew more significant. This trend aligns with the logarithmic time complexity of the binary heap's insertion operation, highlighting the expected performance behavior. Despite the increase in average insertion times, the measured values remained relatively small, even for the most significant input size 100000. This demonstrates the efficiency of the binary heap data structure for fast insertion of elements, making it a suitable choice for scenarios where efficient insertion operations are required. Future research could investigate other operations, such as deletion, to comprehensively analyze the binary heap's performance.

**References**

1. Bozhinovski, A., Tanev, G., Stojchevska, B., Pachovski, V., & Ackovska, N. (2016). Time complexity analysis of the binary tree roll algorithm. International Journal of Computer Applications.
2. Jindal, P., Kumar, A., & Kumar, S. (2010). Analysis of time complexity in a binary search tree. In Proceedings of the 4th National Conference; INDIACom.
3. Papadimitriou, C. H. (2003). Computational complexity. In Encyclopedia of computer science (pp. 260-265