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

**Institution affiliation**

**Student’s Name**

**Course Name & Number**

**Instructor’s Name**

**Date of Submission**

Table of Contents

[Abstract 3](#_Toc140672772)

[Introduction 4](#_Toc140672773)

[Binary Heap data structure 4](#_Toc140672774)

[Working of Binary heap 5](#_Toc140672775)

[Implementation of binary heap 6](#_Toc140672776)

[a. Used Functions 6](#_Toc140672777)

[b. Total operations inside the loop 7](#_Toc140672778)

[Time complexity Analysis 7](#_Toc140672779)

[i. Insertion Best Case 7](#_Toc140672780)

[ii. Insertion Average Case 7](#_Toc140672781)

[iii. Insertion Worst case 8](#_Toc140672782)

[Uses of Binary heap 8](#_Toc140672783)

[Conclusion: 9](#_Toc140672784)

[References 10](#_Toc140672785)

[Screenshots 11](#_Toc140672786)

# Acknowledge

# I sincerely wish to take this privilege to thank the Almighty God for His love, care, divine protection, guidance, good health and the opportunity of a memorial and fruitful period during my project study.

# 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. 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

Data structures and algorithms are fundamental pillars of computer science, indispensable for solving a wide range of computational problems efficiently. Data structures offer systematic ways to organize and store data, encompassing arrays, linked lists, stacks, queues, trees, and graphs. On the extra hand, algorithms offer step-by-step instructions for carrying out specific responsibilities on the data. Selecting the right data structure and augmenting algorithms is critical, as it significantly impacts the performance and scalability of software applications.

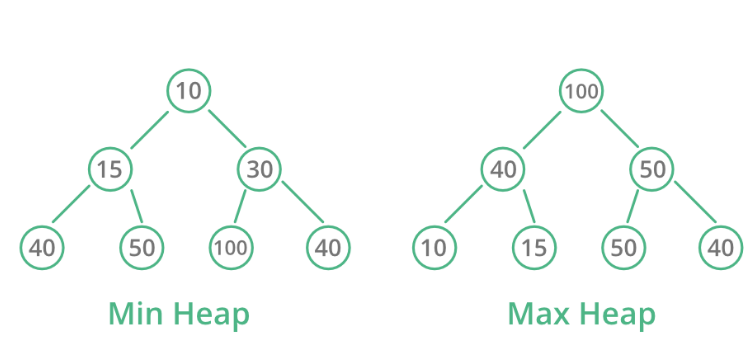
The efficacy of data structures and algorithms is measured through time and space complications, which define how quickly an algorithm finalizes a task and how much memory it requires. Understanding these concepts is vital for programmers, software engineers, and developers, permitting them to enterprise efficient solutions, optimize code, and create high-performance presentations.

This project examines the binary heap data structure's typical case insertion time. The effectiveness of the binary heap was resolute by measuring and comparing the insertion time for various input sizes. A binary heap application that enables the dimension of insertion times was built for this research. The results of trials with various input sizes were studied to learn more about the typical case supplement time for binary heaps.

# Binary Heap figures structure

A Binary pile is a characteristic **Tree-based Data Structure** in which the tree is a [complete tree](https://www.geeksforgeeks.org/complete-binary-tree/). A second pile is typically understood using an array, where the elements are ordered in a way that gratifies the heap property. The root component is stored at index 0, and for any component at index "i," its left child is positioned at index "2i + 1," and its right child is situated at index "2i + 2." A binary heap is a binary tree-based facts structure that satisfies the heap stuff

1. **Min-Heap Assets**: For each node "i," the worth of "i" is less than or equivalent to the values of its children. In other arguments, the minutest element is continually at the root, and the price of any parent node is less than or equivalent to its children's tenets.
2. **Max-Heap Assets**: For each node "i," the charge of "i" is greater than or alike to the standards of its children. In other difference of opinion, the maximum element is always at the root, and the worth of any parent node is greater than or equal to its children's value



**Operations in a binary heap**

1. **Heapify:** a method of making a heap from an assortment.
2. **Insertion:** procedure of supplementing an element in prevailing heap
3. **Deletion:** erasing the top section of the heap or the highest significance element, and then organizing the heap and recurring the element.
4. **Peek:** to check first (or can say the top) element of the heap.

# Working of Binary heap

This project implements a min heap data structure, focusing on only insertion operation. The program generates random elements and then inserts them into the heap. The insertion of elements follows a specific process to maintain the min-heap property.

The following processes are performed when inserting an element into the min-heap

1. A new element is appended to the end of the heap. This maintains the complete binary tree property of the heap.
2. A "sift-up" operation is performed on the newly inserted element. This operation ensures that the min-heap property is preserved.
   * The worth of the newly inserted section is compared with its parent node's value.
   * If the worth of the parent bulge is greater than the value of the inserted element, swap operation is performed.
   * This process is repeated until either the element reaches the root of the heap or the min-heap property is satisfied (the parent's value is smaller).

The sift-up operation guarantees that the new element "bubbles up" the heap until it finds its appropriate position where the min-heap property is maintained.

1. After the insertion and the sift-up operation, the new element is successfully inserted into the min-heap while preserving the min-heap property.

# 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.

The following functions were defined:

## Used Functions

1. generate\_input\_data(size): This function generates a list of random integers within a given size range.
2. measure\_average\_insertion\_time(size, num\_iterations): This function measures the average insertion time for a given input size and number of iterations.
3. perform\_experiments(input\_sizes, num\_iterations): This function performs experiments for different input sizes and records the average insertion times.
4. create\_ui(results): This function creates the user interface using tkinter to display buttons, labels, and a table for the experiment results.
5. on\_button\_click(size): This function handles button clicks, initiates the insertion time measurement, and updates the UI accordingly.
6. display\_scatter\_plot(): This function displays a scatter plot of the experiment

## b. Total operations inside the loop

The primary loop with the most operations is in the measure\_average\_insertion\_time function. Inside this loop, for each iteration, the function generates a list of random integers, creates a new binary heap, inserts each element into the heap, and calculates the insertion time. The number of operations inside this loop depends on the input size and the number of iterations.

**c. Total operations**

The total operations in the code depend on the number of iterations, the input sizes, and the size of the input data. For each input size, the code performs num\_iterations iterations of insertion and time measurement.

# Time complexity Analysis

## Insertion Best Case

The best scenario for introducing an element into a min-heap occurs when the value being inserted is greater than its parent's value, here no additional checks or shifts are required. The actions performed is limited to one check and no change the time complexity for this case turns out to be O(1)

## Insertion Average Case

When inserting an element into a heap, there are different possible scenarios depending on the value of the element being inserted and its relationship with its parent:

**Best Case:** The best-case scenario occurs when the element being inserted is larger (in the case of a MinHeap) or smaller (in the case of a MaxHeap) than its parent. In this condition, we only require to complete a single check to safeguard that the heap possessions is satisfied, and no further shifts or swaps are compulsory. This best-case scenario has a time complexity of O(1) because it involves only a constant number of operations.

**Second Best Case**: The second-best case occurs when the element being inserted needs to be moved up the heap to satisfy the heap property. This requires checking the element's parent, shifting the element upwards if needed, and then repeating the process with its new parent until the element reaches the correct position. The number of checks and shifts increases as we move up the heap towards the root. The total number of checks and shifts required is logarithmic, with a time complication of O(log N), where N is the figure of components in the heap. This scenario is more likely to happen as the heap grows larger.

That is,

1, 2, 3, 4, 5, ……., logN

The average would be,

O(Average)=

O(Average)=O((logN(logN+1)2)logN)

O(Average)=O(logN+12)

Therefore;

O(Average)=O(logN)

## Insertion Worst case

In the worst-case scenario of inserting a value into a Min-heap, the value will have to be shifted all the way to the root of the heap. This occurs when the inserted value is either the greatest (for MaxHeap) or the smallest (for MinHeap).

To range the root, we need to navigate through one parent node on every level of the heap. Since the height of a heap is determined by log(N), where N is the number of features in the heap, we will have log(N) levels to traverse.

Consequently, the time difficulty for the worst situation will be O(logN)

# Uses of Binary heap

1. Priority Queues: Binary lots are commonly used to device priority queues, where elements with higher priority are dequeued before basics with lower priority. Priority queues find presentations in task scheduling, event handling, Dijkstra's shortest path algorithm, and Huffman coding.
2. File Density: Heaps are cast-off in data compression procedures such as Huffman coding, which uses a importance queue implemented as a min-heap to form a Huffman tree.
3. Dynamic software design: Heaps are used in dynamic programming procedures such as the greedy algorithm, where fundamentals are ordered in order of importance.
4. Network Routing: In network steering algorithms, binary heaps can be used to preserve a priority queue of nodes or routes, portion to optimize the pathfinding procedure

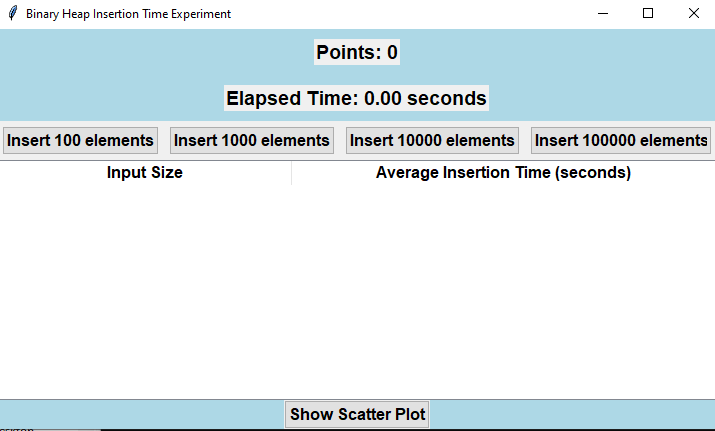
# Conclusion:

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.

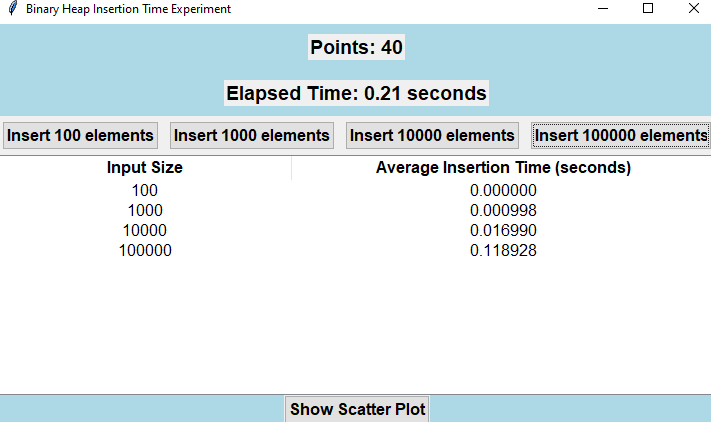
# References

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

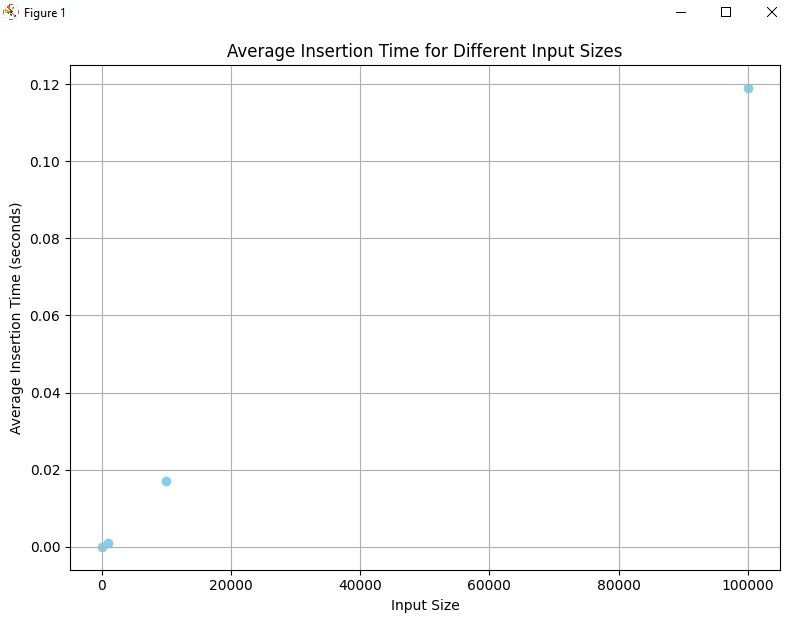
# Screenshots



A screenshot showing the program after it has been run



Screenshot showing the insertion times after insert buttons are clicked



Screen shot showing a scatter plot graph of average insertion times against the input sizes