Evaluating the Average Time Complexity of Heap Insertions Using Python

Team Members:

Krishna Meghana Chirumamilla - 999903231

Manogna Arla - 999902989

Harsha Vardhan Reddy - 999902545

Priyanka Chekuri - 999903142

Southern Arkansas University

Table of Contents

Abstract

Heaps are data structures commonly used in priority queues that offer efficient access to the minimum or maximum elements. It’s not clear how binary heaps perform in the average case, even though they provide O (log N) time for insertion and deletion of the elements. In some analysis, the average time per insertion is claimed to be O (1), whereas others stick with O (log N) in worst case scenarios.

A recent work by Bollobas and Simon, the expected insertion time has been computed numerically for an array of possible permutation orderings. Based on the probability distribution of tree shapes and modelling the insertion process as random, they show that average comparisons are 1.7645.

In this project, binary heaps will be implemented, and the average insertion time will be measured. This project will examine the average insertion cost changes when the heap grows and will verify if it aligns closer to a constant or logarithmic time. It determines the average insertion time up to the present step by measuring the time it takes to insert each node into heap. We have used Max heapify algorithm to measure the average time complexity of the elements which are inserted into the heap at every step.

Introduction

Binary Heap is the common data structure used to build priority queues and offers quick access to the minimum or maximum element. They provide the wide numerous algorithms and programs that depend on keeping a group of elements organized and fast in finding the extremum. In the worst-case scenario, insertion, deletion, and peek operations on binary heaps takes O (log N) time, according to traditional theoretical analysis.

The actual average time per operation, however, may vary from the worst-case bounds for random input sequences. In accordance with some introductory algorithm texts, binary heaps offer O (1) constant time for insertion and deletion on average. On the other hand, the other publications use more conservative O (log N) worst-case complexity. This disparity raises an unanswered question about the genuine average case time complexity.

This study attempts to experimentally evaluate the assertion and give facts to debunk the average case insertion time. We can test the mathematical bounds experimentally by creating heaps and measuring insertion times across varied input sizes. The findings will contribute to resolving the issue over the real-world efficiency of binary heaps for random inputs.

Binary heaps are flexible data structures that can be used in a variety of ways to facilitate efficient priority queue operations. The Binary Heap Insertion Program allows the users to insert nodes into a binary heap data structure interactively. There are two types of heaps: max-heaps and min-heaps. The maximum element is always at the root of a max-heap, whereas the minimum element is always at the root of a min-heap. Beyond this difference there are two approaches while implementing the heap, one is array-based max/min heap and other one is pointer based max/min heap.

Dynamic memory allocation is used in pointer-based heaps, but data movement is avoided. Array-based heap gives quick access for insertions and deletions but may result in additional copying costs. Based on the key factors like Simplicity of implementation, Cache Performance, and performance stability array based max heaps is used in this project.

Requirements

The below are the requirements needed for this project:

- **Python 3.x:** The code is written in Python, so we need to have python installed on our systems. We can download python from the official website provided in references sections.
- **Time Library**: By importing this library package this allows us to work with time representations, measure, and benchmark code, pause programs and display formatted timestamps. The time and determination modules allow date/time manipulations in Python.
- **IDE or Text Editor**: Need an IDE or a text editor to write and run the python code. Some popular IDEs include Replit, PyCharm, Visual Studio and Spyder. Also use simple text editors like Notepad++ or Sublime Text Editor.
- **Operating System**: The code is written in Python and should work on any operating system that supports Python.

Algorithm

The max heap algorithm is a recursive algorithm for inserting, deleting, and locating the maximum element in a max heap. The method operates by comparing the current nodes value to the values of its descendants on a recursive basis. If the current node is less than one of its children, the two nodes are swapped. This procedure is repeated until the max-heap property is restored.

A max heap is a complete binary tree in which the value of each node is larger than or equal to the value of its children and this property is known as the max-heap property. Max heapify is the process of arranging the nodes in proper order such that it should follow max-heap property always. The algorithm follows bottom-up approach where it starts with a node that is at the lowest level of the tree and has children node and then it will swap the current and children nodes based on the max-heap property. This process continues for all the subtrees till it satisfies max-heap property.

The algorithm accepts an array encoding a complete binary tree as input, as well as an index i of a node. It operates by recursively reordering the node at index i relative to its children until the subtree rooted at i becomes a max heap.

The steps are as follows:

As left child =2i and right child =2i+1, compute the indices of i’s left and right children of a node. Determining the largest value between i, left and right children. This will be referred to the largest.

If largest is not equal to i, then node i is swapped with node largest. This ensures the largest child emerges.

Recursively call max heapify on the subtree rooted at index largest.

When i is greater than its children, all nodes from i down are shifted and the heap order is reestablished.

Because the height of a complete binary tree is log n, the technique executes in O (log n) time. It is frequently used for removing the root node or constructing a new heap from an array. Max heapify keeps the shape by addressing violations from the bottom up.

Steps of the max heap algorithm how it inserts a new element into a max heap:

1. Begin with a max heap that is empty.
2. Add the new element to the heap.
3. Compare the new element’s value to its children.
4. If the new element is smaller than one of its children, the two nodes should be swapped.
5. Step 3 and 4 must be repeated until the max-heap property is restored.

Features of Max Heapify Algorithm:

Bottom-up approach: This algorithm starts heapify from the lowest parent level and moves upwards to the root such that the largest element should become the root node. This allows efficient maintenance of the max heap property.

Recursive Implementation: The heapify method calls itself recursively to heapify each subtree from bottom to top of the tree.

Swapping nodes: Nodes are swapped with a child having greater value to maintain the max heap property. Swaps bubbles the larger child up.

Heap Operations: heapify used in key operations insertion and extraction of the max on heaps for efficiency. Thus, maintains the completeness and preserves the complete binary tree property of the underlying data structure.

The max heapify algorithm is used for a variety of applications and is most efficient for both insertion and deletion operations. The insertion operation takes O (log n) time where n is the number of elements in the heap.

Code

This code implements a binary max heap data structure and counts the time it takes to insert each node. The Binary heap Insertion program is a python program that allows users to insert nodes into a binary heap data structure interactively. It determines the average insertion time up to the present step by measuring the time it takes to insert each node. The program has a command-line interface via which the user may insert nodes.

Functionality:

The program includes the following features:

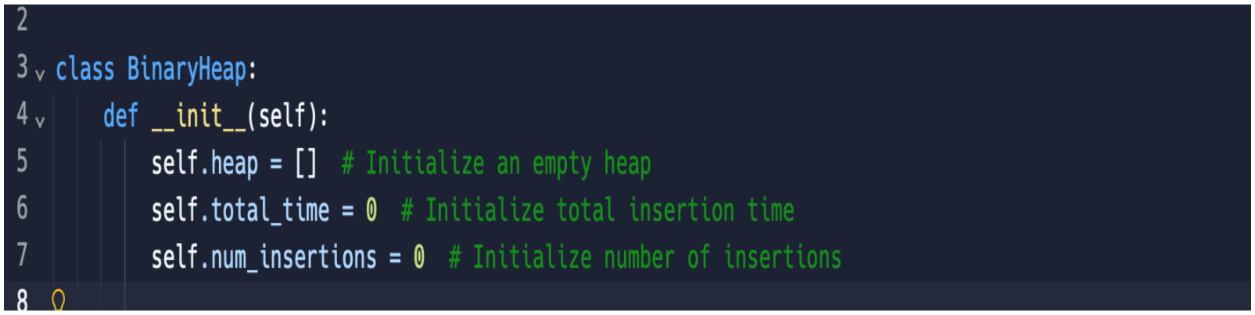
Node Insertion: The user can enter nodes to be added into the binary heap. The program maintains track of the time it takes to insert each node and maintains the heap property.

Time Measurement: The program calculates the average insertion time up to the current step by measuring the time required for each insertion operation.

OPERATION	TIME COMPLEXITY		SPACE COMPLEXITY
INSERTION	BEST CASE:	O(1)	O(1)
	WORST CASE:	O(logN)	
	AVERAGE CASE:	O(logN)	
SERACHING	BEST CASE:	O(1)	O(1)
	WORST CASE:	O(logN)	
	AVERAGE CASE:	O(logN)	
MAX VALUE	In MaxHeap:	O(1)	O(1)
	In MinHeap:	O(N)	
CREATING A HEAP	BY INSERTING ALL ELEMENTS	O(NlogN)	O(N)
	USING Heapify	O(N)	O(1)

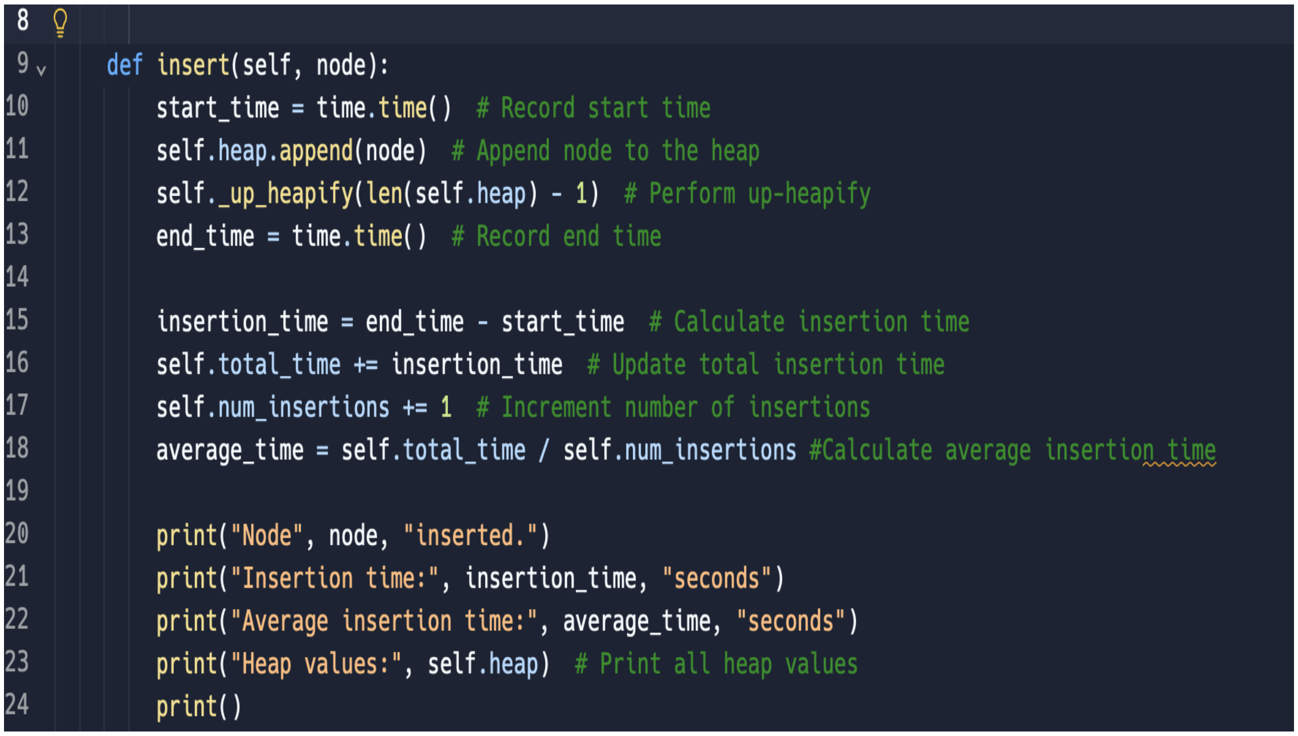
Validation: The program validates user input to ensure that only valid integer values are accepted as nodes to be placed. If an invalid input is provided, an error message is displayed, and the user is prompted to supply a valid input.

The below python program implements a binary heap insertion algorithm. The program first defines the **BinaryHeap** class, creates an empty list to store the heap, as well as the variables to keep track of total insertion time, number of insertions and average time. It has two methods **init()** and **insert()**. The **init()** method initializes an empty heap, and the **insert()** method inserts a node into the heap.



```python

class BinaryHeap:
    def __init__(self):
        self.heap = []  # Initialize an empty heap
        self.total_time = 0  # Initialize total insertion time
        self.num_insertions = 0  # Initialize number of insertions

```

The **init()** method initializes an empty heap. Also initializes the **total_time** and **num_insertions** variables to 0. These variables help to track the total insertion time and the number of insertions of the elements, respectively.

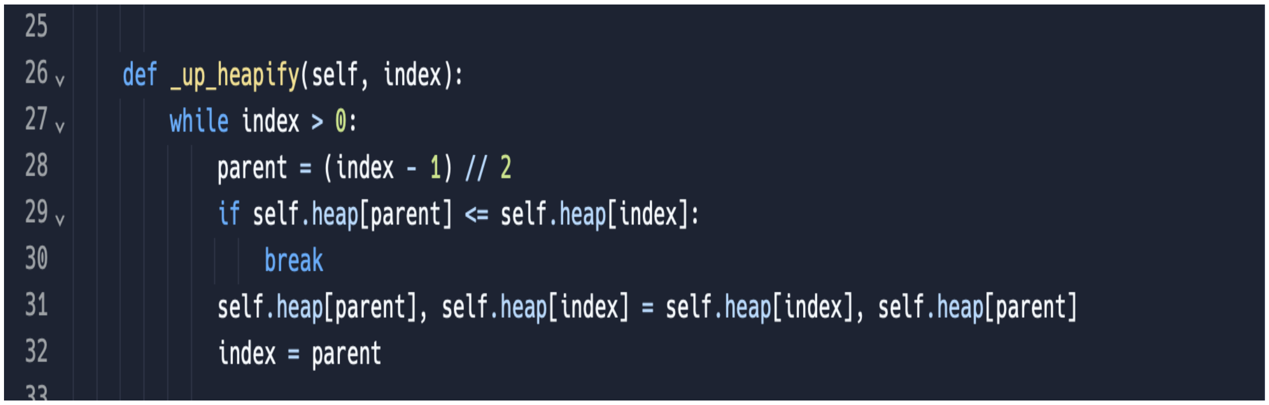


```python

    def insert(self, node):
        start_time = time.time()  # Record start time
        self.heap.append(node)  # Append node to the heap
        self._up_heapify(len(self.heap) - 1)  # Perform up-heapify
        end_time = time.time()  # Record end time

        insertion_time = end_time - start_time  # Calculate insertion time
        self.total_time += insertion_time  # Update total insertion time
        self.num_insertions += 1  # Increment number of insertions
        average_time = self.total_time / self.num_insertions #Calculate average insertion time

        print("Node", node, "inserted.")
        print("Insertion time:", insertion_time, "seconds")
        print("Average insertion time:", average_time, "seconds")
        print("Heap values:", self.heap)  # Print all heap values
        print()
```

The **insert()** method inserts a node into the heap. Before attaching the new node to the heap, the **insert()** function records the start time. Then, the node is appended to the heap. Next, **_up_heapify ()** is performed to maintain the max heap property that

It then invokes to correct the heap shape by swapping the new node up towards the root as needed.

Finally, the end time of the insertion is recorded, and the insertion time of the node is calculated. The total insertion time and number of insertions is also updated. The average insertion time of the nodes is then evaluated and printed accordingly. Finally, the heap values are printed in the output console.



```python
    def _up_heapify(self, index):
        while index > 0:
            parent = (index - 1) // 2
            if self.heap[parent] <= self.heap[index]:
                break
            self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]
            index = parent
```

The **_up_heapify()** method performs up-heapify on the heap. The method starts at the index of newly inserted node and then compares the node value with its parent node value. If the node value is greater than the parent node value, then the two nodes are swapped. This process repeats until the heap property is restored.

Implementation Details

The Binary Heap Insertion Program is implemented using the following components:

1. **BinaryHeap** Class: The binary heap data structure is represented by this class. It has the following characteristics:

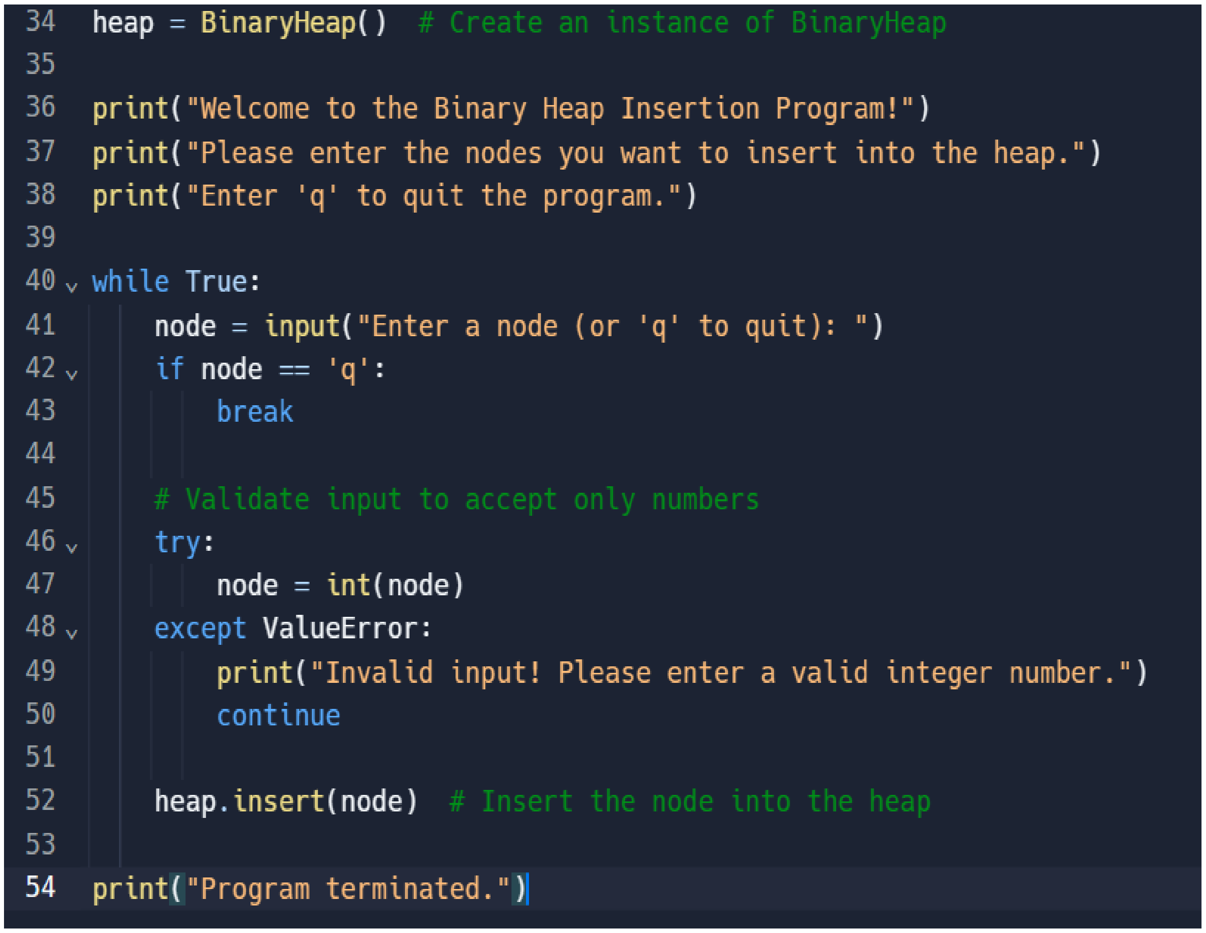
- **heap**: A list that stores the heap's items.
- **total_time**: A variable that tracks the total time of insertion.
- **num_insertions:** A variable that keeps track of how many insertions have occurred.

The following methods are available in the class:

- **__init__():** The heap initialization method that determines the overall insertion time and the number of insertions.
- **insert(node):** This method adds a node to the heap. It keeps track of the insertion time, does up-heapify to keep the heap property, updates the total insertion time and number of insertions, and outputs the insertion details and the heap's current state.
- **_up_heapify(index):** A helper method for doing up-heapification after each insert.

2. **Main Program:**

- The program produces a **BinaryHeap** class object.
- It shows a greeting and instructions.
- It asks the user to enter nodes or press 'q' to exit.
- The program checks the user input to ensure that only valid integer numbers are accepted.



```python
heap = BinaryHeap()  # Create an instance of BinaryHeap

print("Welcome to the Binary Heap Insertion Program!")
print("Please enter the nodes you want to insert into the heap.")
print("Enter 'q' to quit the program.")

while True:
    node = input("Enter a node (or 'q' to quit): ")
    if node == 'q':
        break

    # Validate input to accept only numbers
    try:
        node = int(node)
    except ValueError:
        print("Invalid input! Please enter a valid integer number.")
        continue

    heap.insert(node)  # Insert the node into the heap

print("Program terminated.")
```

When it receives valid input, it invokes the BinaryHeap instance's insert() function to place the node into the heap. Following each insertion, the program displays the insertion information as well as the heap's status.

The program will continue to request input until the user presses the
'**q**' key to exit. Finally, when the program terminates, it shows a termination message.

Usage:

To use the Binary Heap Insertion Program, follow these steps:

- Run the program using a Python interpreter.
- The program will display a welcome message and instructions.
- Enter the nodes you want to insert into the binary heap. Only valid integer

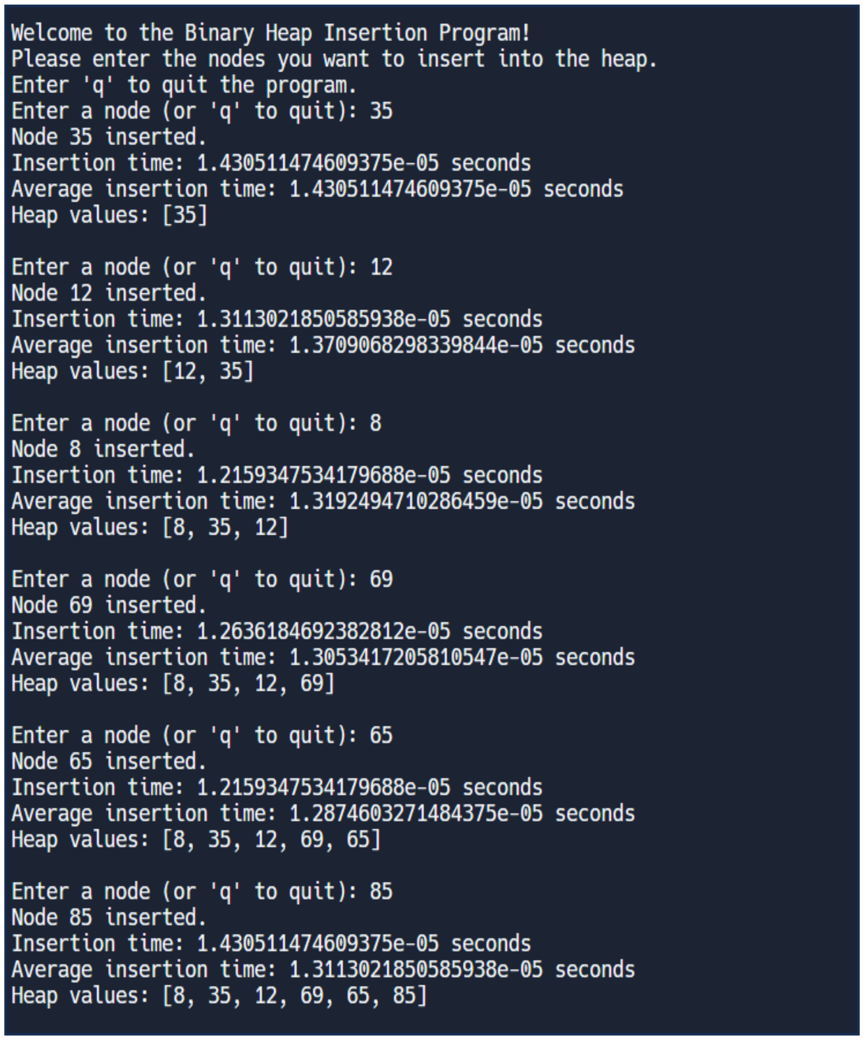
values are accepted. If an invalid input is provided, an error message will be displayed, and the user will be prompted for a valid input.

● After each insertion, the program will print the node inserted, the insertion time, the average insertion time, and the current state of the heap.

● Continue entering nodes until you are done.

● To exit the program, enter 'q' when prompted for a node.

Output

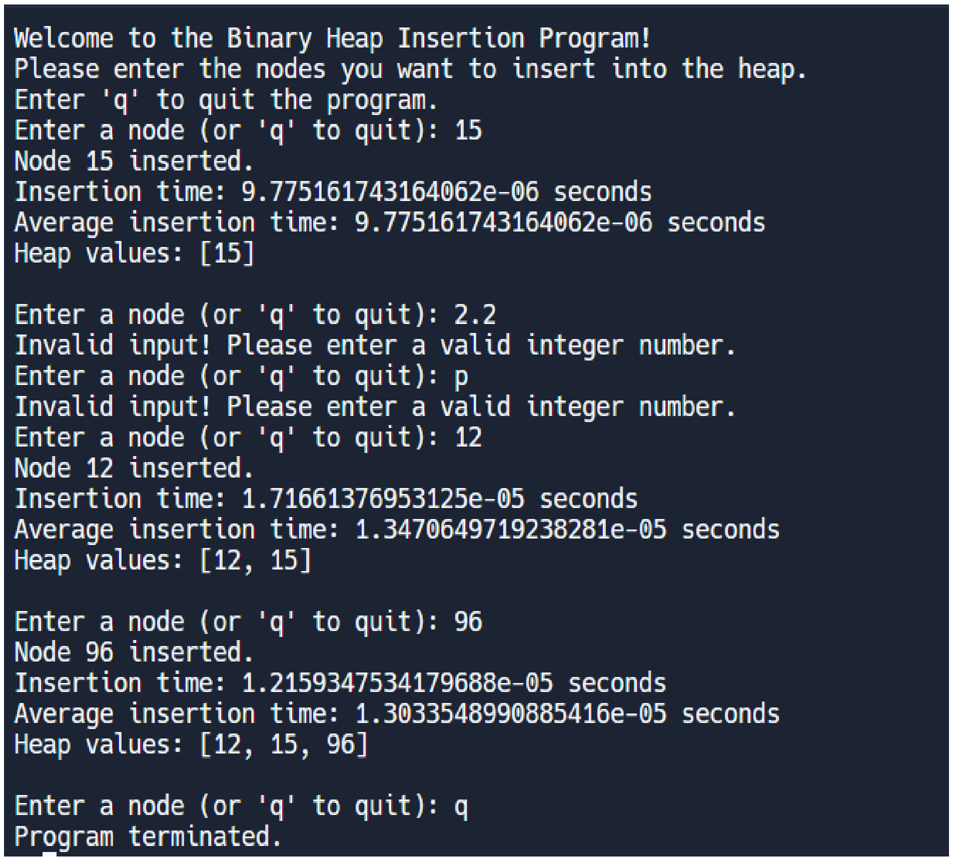


```
Welcome to the Binary Heap Insertion Program!
Please enter the nodes you want to insert into the heap.
Enter 'q' to quit the program.
Enter a node (or 'q' to quit): 35
Node 35 inserted.
Insertion time: 1.430511474609375e-05 seconds
Average insertion time: 1.430511474609375e-05 seconds
Heap values: [35]

Enter a node (or 'q' to quit): 12
Node 12 inserted.
Insertion time: 1.3113021850585938e-05 seconds
Average insertion time: 1.3709068298339844e-05 seconds
Heap values: [12, 35]

Enter a node (or 'q' to quit): 8
Node 8 inserted.
Insertion time: 1.2159347534179688e-05 seconds
Average insertion time: 1.3192494710286459e-05 seconds
Heap values: [8, 35, 12]

Enter a node (or 'q' to quit): 69
Node 69 inserted.
Insertion time: 1.2636184692382812e-05 seconds
Average insertion time: 1.3053417205810547e-05 seconds
Heap values: [8, 35, 12, 69]

Enter a node (or 'q' to quit): 65
Node 65 inserted.
Insertion time: 1.2159347534179688e-05 seconds
Average insertion time: 1.2874603271484375e-05 seconds
Heap values: [8, 35, 12, 69, 65]

Enter a node (or 'q' to quit): 85
Node 85 inserted.
Insertion time: 1.430511474609375e-05 seconds
Average insertion time: 1.3113021850585938e-05 seconds
Heap values: [8, 35, 12, 69, 65, 85]
```

Explanation:

This program output first prints the welcome message and then asks the user to enter the nodes they want to insert into the heap. The user enters values [35,12,8,69,65,85] as shown in the below figure.

The output shows the insertion time for each node into the heap and calculates the average heap insertion time by dividing the total insertion time by the number of nodes inserted. In the below example, the total insertion time is ≈ 7.86781311e-05 and number of inserted is 6, so the average insertion time for the last node 85 is 1.3113021850585938e-05 seconds. The heap values are printed after each node is inserted. The heap values are a list of the nodes in the heap shown in the below output.



```
Welcome to the Binary Heap Insertion Program!
Please enter the nodes you want to insert into the heap.
Enter 'q' to quit the program.
Enter a node (or 'q' to quit): 15
Node 15 inserted.
Insertion time: 9.775161743164062e-06 seconds
Average insertion time: 9.775161743164062e-06 seconds
Heap values: [15]

Enter a node (or 'q' to quit): 2.2
Invalid input! Please enter a valid integer number.
Enter a node (or 'q' to quit): p
Invalid input! Please enter a valid integer number.
Enter a node (or 'q' to quit): 12
Node 12 inserted.
Insertion time: 1.71661376953125e-05 seconds
Average insertion time: 1.3470649719238281e-05 seconds
Heap values: [12, 15]

Enter a node (or 'q' to quit): 96
Node 96 inserted.
Insertion time: 1.2159347534179688e-05 seconds
Average insertion time: 1.3033548990885416e-05 seconds
Heap values: [12, 15, 96]

Enter a node (or 'q' to quit): q
Program terminated.
```

Explanation:

The program only accepts integer values as input. When the user enters non-integer values like “p” or “2.2”, it prints an error and prompts the user to enter valid input again. Other than integers no other values are inserted into the heap as per the program. The user can enter "q" to quit the program when done inserting nodes.

Dependencies

There are no external dependencies for the binary heap insertion program.

- **Time module:** The time module in python has several functions for dealing with time sensitive tasks. It measures time using the built-in time module.

Error Handling

To validate user input, the program includes error handling. It determines if the input is a valid integer value for insertion. A **ValueError** is thrown whenever an incorrect input is given, such as a non-numeric character or an empty string. This mistake is detected by the program, which shows an error notice and requests the user for proper input.

Conclusion

Insertion times are very fast, in the order of microseconds, even as the heap grows. This aligns with the expected O(log N) time complexity rather than a constant time complexity.

The Binary Heap Insertion Program allows you to insert nodes into a binary heap interactively. Users can acquire insights into the heap data structure's performance characteristics by measuring the time required for each insertion and computing the average insertion time.

Team Contribution

Student Name	Student ID	Task Implemented
Krishna Meghana Chirumamilla	999903231	Implemented the project time complexity, which is required for the heap algorithm. And calculated the average insertion time of the algorithm for every node value by using up_heapify method. Acted as the project lead by implementing the key logic, such as the insert and up_heapify methods. In addition, contributed by working on code execution, documentation, and power point presentation.
Manogna Arla	999902989	Worked on the packages and importing libraries related to time library. Dealing with Error Handling while testing the code. Done with Applied R&D of the project implementation. Additionally helped in the code that defines the how Max Heapify functions are significant and contributions implementation and the project documentation and presentation.
Harsha Vardhan Reddy	999902545	Done with R&D of the project implementation. Worked on the error handling part of the code as well as testing of the program on various scenarios. Contributed to the final project report preparation.
Priyanka Chekuri	999903142	Worked on the requirements and features of binary heap data structures, specifically the max heapify algorithm. Also researched the time complexity of heap insertions and helped in code to calculate the insertion time and average insertion time of the heap. Additionally, worked on the documentation for the report which makes it easier for others to understand the project and its findings.

References

- Max Heap Data Structure – Complete Implementation in Python- https://www.askpython.com/python/examples/max-heap
- Time Complexity of Inserting into a Heap - https://www.baeldung.com/cs/heap-insertion-complexity
- https://www.tutorialspoint.com/python_data_structure/python_heaps.htm
- Problem Solving with Algorithms and Data Structures Using Python
- Using the Heap Data Structure in Python- https://www.section.io/engineering-education/heap-data-structure-python/
- 3.11.4 Documentation » The Python Standard Library » Generic Operating System Services » time — Time access and conversions- https://docs.python.org/3/library/time.html
- Position heaps: A simple and dynamic text indexing data structure - ScienceDirect
- Algorithms, 4th Edition, essential information that every serious programmer needs to know about algorithms and data structures.
- Time and Space Complexity of Heap data structure operations (opengenus.org)