# Hybrid Priority queue

# Implemented using combining min-heap and hash-map

GROUP NAME : **DSA DYNAMOS**

GROUP MEMBERS :

|  |  |
| --- | --- |
| NAME | ROLL NUMBER |
| DASARI SAI SAMRAT | CB.EN.U4CSE21415 |
| GUDIVADA ROHAN LAL | CB.EN.U4CSE21420 |
| KARRI AMMI REDDY | CB.EN.U4CSE21428 |
| SAGIRAJU HARINADHA RAJU | CB.EN.U4CSE21451 |
| VINAYAKAN V S | CB.EN.U4CSE21467 |

Introduction :

Hybrid data structures refer to the combination or integration of multiple data structures to create a more efficient and specialized structure for solving complex problems. These structures leverage the strengths of different data structures to overcome their individual limitations, resulting in improved performance and optimized solutions.

The significance of hybrid data structures lies in their ability to address the specific requirements of complex problems that cannot be efficiently solved using a single data structure. While single data structures have their own strengths, they may not be optimized for all aspects of a problem, such as time complexity, space efficiency, or specialized functionalities. By combining multiple data structures, developers can achieve a balance between these factors and create a more powerful solution.

Here are a few reasons why hybrid data structures can be better than single data structures:

1. Enhanced Performance: Hybrid data structures can offer superior performance compared to single data structures by leveraging the strengths of each component. For example, a combination of a hash table and a min-heap can provide both fast search operations (hash table) and priority findings(min-heap) resulting in overall improved performance.

2. Tailored Functionality: Complex problems often require specialized functionalities that may not be available in a single data structure. Hybrid data structures allow developers to combine different structures to create a solution that addresses specific requirements.

3. Flexibility and Adaptability: Complex problems may have dynamic characteristics, with changing requirements or data patterns. Hybrid data structures can be more flexible and adaptable to these changes by allowing components to be modified or replaced based on evolving needs. This flexibility enables efficient problem-solving in dynamic environments.

4. Trade-off Resolution: Different data structures often involve trade-offs between factors such as time complexity, space efficiency, and ease of implementation. Hybrid data structures provide an opportunity to strike a balance between these trade-offs. By carefully selecting and combining components, developers can optimize the structure to achieve the best compromise for a given problem.

In summary, hybrid data structures offer improved performance, tailored functionality, optimized space utilization, flexibility, adaptability, and resolution of trade-offs. They can efficiently solve complex problems that cannot be efficiently addressed using a single data structure alone. However, designing and implementing hybrid data structures require careful consideration of trade-offs and understanding the problem requirements to harness their full potential.

Overview of the Hybrid Data Structure:

Hybrid Priority Queue, combines multiple data structures, typically a min-heap and a hash map, to solve specific problems efficiently. The composition of the hybrid priority queue includes:

1. Min-Heap: A min-heap is a binary tree-based data structure where the parent node has a lower value than its child nodes. In the context of a priority queue, the min-heap ensures that the element with the highest priority (lowest value) is always at the root, allowing for efficient retrieval and removal of the highest-priority element.
2. Hash Map: A hash map, also known as a hash table, is a data structure that maps keys to values using a hash function. It provides efficient key-value pair lookups, insertions, and deletions. In the context of a hybrid priority queue, the hash map is used to associate elements with their unique identifiers, enabling fast access, update, and removal based on identifiers.

The advantages and motivations behind using a hybrid data structure like the Hybrid Priority Queue are:

1. Efficient Priority-Based Operations: The hybrid priority queue allows for efficient operations based on priority, such as retrieving the highest-priority element, updating an element's priority, or removing elements based on their priority. The min-heap ensures the highest-priority element is readily accessible, while the hash map enables quick identification and modification of elements by their identifiers.
2. Fast Lookup and Update by Identifier: The hash map component of the hybrid priority queue enables fast lookup and update operations based on element identifiers. This is particularly useful when you need to locate and modify specific elements in the priority queue efficiently.
3. Flexibility and Versatility: The hybrid nature of the data structure provides flexibility and versatility in handling different types of problems. It combines the strengths of both the min-heap and the hash map to optimize specific operations, making it suitable for various scenarios that require efficient priority-based operations with fast lookup and update capabilities.
4. Scalability: The hybrid priority queue can scale well with a large number of elements. The min-heap ensures efficient retrieval and removal of the highest-priority element, while the hash map provides constant-time lookups and updates based on identifiers. This scalability is important in scenarios where the number of elements and the frequency of operations are significant factors.

In conclusion, the Hybrid Priority Queue offers a powerful solution for efficiently managing elements with priorities while allowing fast lookup and update operations. By combining the strengths of a min-heap and a hash map, it provides an effective and versatile data structure for solving specific problems that involve priority-based operations and efficient element identification.

Top of Form

Implementation Details:

Implementation Process:

* + The HybridPriorityQueue class is defined with two main attributes: 'heap' and 'map'. The 'heap' stores the element details, and the 'map' is used as a hash map to map priorities to their corresponding indices in the heap.
  + The hash\_function method is implemented to calculate the index in the map based on the priority value.
  + The insert method is used to insert elements into the priority queue. It calculates the index using the hash\_function and handles collisions by linearly probing through the map until an empty slot is found.
  + The bubbleUp method is responsible for maintaining the min heap property by bubbling up the newly inserted element to its correct position.
  + The swap method is used to swap elements in the heap and update the corresponding indices in the map.
  + The displayHeap method returns the heap, and the displayHash method prints the contents of the map.
  + The search method allows searching for an element based on its priority. It uses the hash\_function to calculate the index and linearly probes through the map until the element is found or an empty slot is encountered.
  + The delete method deletes an element from the priority queue based on its priority. It uses the search method to find the element's index and then swaps it with the last element, removes it from the heap, and adjusts the heap using bubbleUp and bubbleDown operations.
  + In hash map it stores a node which contains Priority as Key, Index position of Priority in Min-heap as Value.

Design Choices and Trade-offs:

* + The choice of a min heap as the underlying data structure for prioritization ensures efficient insertion, deletion, and retrieval of the minimum priority element in O(log n) time complexity.
  + The use of a hash map provides efficient name-based operations. However, collisions are handled using linear probing, which may result in longer search times if many collisions occur.
  + The implementation uses an array-based approach to store the heap and map, which simplifies indexing and memory management but can lead to potential space wastage if the data set is sparse.
  + The hash function uses modular arithmetic to map priorities to indices in the map. This approach assumes a fixed-size map (31 in this case) and may not provide optimal distribution of elements if the priorities are not evenly distributed.

In summary, the implementation integrates the min heap and hash map to create a hybrid data structure that combines efficient prioritization with fast name-based operations. The design choices made aim to balance performance and functionality, although there may be trade-offs such as potential collisions and suboptimal space utilization.

CODE LINK : <https://github.com/RohanlalGudivada/DSA-DYNAMOS/commit/1ebe004127d997e94e70a2bfb9d729c562ec4bc0>

Practical Applications:

USE CASE : HOSPITAL MANAGEMENT SYSTEM

This is a Scenario of a Hospital Management system where this Hybrid data structure is used for assigning priorities to the patients based on their respective health problems. The priorities are assigned by the receptionist based on how fast the treatment is required to the patient. The patients goes to the doctor one by one based on the priority/order mentioned by the receptionist. The system takes the information of patients which are given by receptionist. The information contains Priority, Name, Phone Number, Room Number and some more information about the disease. The patients after getting treated by doctor are removed and the system gets updated every-time the patients leave the hospital so that we get to know which patient to be sent next.

For suppose there are some visitors to the hospital to see how the patient is they need to get to know information about patient where the respective patient room has been allotted. In this case there need to be efficient search operation to get to know details of patient here we use a specific data structure where search is faster than any other.

1. Role of MIN-HEAP : This keeps track of the patient’s priority order and keeps on insisting who to go to doctor for the treatment as min-heap is basically used to maintain relation of child and parent where the parent is less value and child is greater value with this analysis we achieve the required efficient calculation who to be next. The min heap allows for efficient extraction of the patient with the highest priority, as well as insertion and deletion operations.
2. Role of Hash-Map : The hash map is used to store the indexes and their corresponding priorities.

This helps us in accessing data in a faster and efficient way the index is stored at the priority index

and through that index we can directly access the heap and fetch the required data efficiently.

This allows for efficient retrieval and modification of patient priorities when needed.

Also this Hybrid Priority Queue can be used in Online ticketing, Efficient URL search, Online delivery system and many more.

Performance Analysis:

Time Complexity Analysis :

CONSIDER ONLY MIN-HEAP:

1. Insertion: The time complexity for inserting a new element into a min heap is O(log n), where n is the number of elements already present in the heap. The algorithm involves placing the new element at the next available position in the heap and then repeatedly comparing it with its parent node and swapping if necessary to maintain the heap property (i.e., the parent node is always smaller than its children).

2. Deletion: The time complexity for deleting the minimum element (root) from a min heap is also O(log n), where n is the number of elements in the heap. The algorithm involves replacing the root with the last element in the heap, reducing the size of the heap by one, and then performing a "heapify" operation to restore the heap property. Heapify compares the new root with its children and swaps with the smaller child if necessary, recursively moving down the tree until the heap property is satisfied.

Both insertion and deletion operations have logarithmic time complexity because the height of a complete binary tree (which a min heap is) is logarithmic with respect to the number of elements. Therefore, these operations are efficient even for large heaps.

3.Search : In a min heap, the search operation does not have an efficient time complexity because the structure of a heap is optimized for efficient insertion and deletion, not searching for specific elements.

To search for a specific element in a min heap, you would typically need to perform a linear search, iterating through all the elements in the heap until you find a match. The worst-case time complexity for searching in a min heap is O(n), where n is the number of elements in the heap.

CONSIDER ONLY HASH-MAP:

1.Insertion (Average case): The average time complexity for inserting an element into a hash map is O(1). It means that on average, the time taken to insert an element is constant, regardless of the size of the hash map.

2. Deletion (Average case): Similar to insertion, the average time complexity for deleting an element from a hash map is also O(1). It takes constant time on average to remove an element.

3. Search (Average case): The average time complexity for searching an element in a hash map is O(1). The hash map uses a hash function to compute the hash value of the search key, which allows for direct access to the corresponding bucket where the element should be located. In an ideal scenario with a good hash function and a low collision rate, the search operation takes constant time on average.

However, it's important to note that in the worst case, the time complexity for all three operations can be O(n), where n is the number of elements in the hash map. This worst-case scenario occurs when there are frequent hash collisions, causing many elements to be stored in the same bucket and requiring a linear search within that bucket.

To mitigate collisions and ensure better performance, hash maps employ techniques such as bucketing, chaining, or open addressing with techniques like linear probing or separate chaining. These techniques help maintain efficient average-case time complexities for insertion, deletion, and search operations.

CONSIDER HYBRID PRIORITY QUEUE(MIN-HEAP + HASH-MAP):

When implementing a hybrid priority queue using a combination of a min heap and a hash map, the time complexities for insertion, deletion, and search are as follows:

1. Insertion: The time complexity for inserting an element into the hybrid priority queue is O(log n) in the average case. The element is first inserted into the min heap, which takes O(log n) time to maintain the heap property. Additionally, the element is inserted into the hash map, which typically has an average-case constant time complexity of O(1).

2. Deletion: The time complexity for deleting the minimum element (root) from the hybrid priority queue is also O(log n) in the average case. The minimum element is removed from the min heap, which takes O(log n) time to adjust the heap structure. Additionally, the element is also deleted from the hash map, which typically has an average-case constant time complexity of O(1).

3. Search: The time complexity for searching for a specific element in the hash map has an average-case constant time complexity of O(1) if the hash function and collision resolution strategy are well-implemented.

Bottom of Form

Space Complexity Analysis :

The space complexity of data structures can vary based on their implementation and the number of elements they store. Here are the typical space complexities for min heap, hash map, and hybrid priority queue using a combination of min heap and hash map:

1. Min Heap: The space complexity of a min heap is O(n), where n is the number of elements stored in the heap. The heap requires an array-based representation to maintain its structure, and it grows in proportion to the number of elements inserted.

2. Hash Map: The space complexity of a hash map depends on the number of key-value pairs stored and the capacity of the underlying array used for bucket storage. In the average case, the space complexity of a hash map is O(n), where n is the number of key-value pairs. However, in the worst case (when there are hash collisions resulting in long bucket chains), the space complexity can increase to O(n + m), where m is the capacity of the underlying array.

3. Hybrid Priority Queue (Min Heap + Hash Map): The space complexity of a hybrid priority queue using a combination of min heap and hash map is typically O(n) due to the space requirements of both data structures. In this implementation, the elements are stored in the min heap, which requires O(n) space. Additionally, the hash map is used for efficient search operations and requires additional space, typically O(n) in the average case. Therefore, the overall space complexity of the hybrid priority queue is O(n).

It's worth noting that the space complexities mentioned above represent the additional space required by the data structures themselves and do not include the space needed to store the actual data elements.

Experimental Evaluation:

While considering the whole code the experiment takes an execution time of **Execution Time:** 0.066109338   
**Execution Memory:** 9524 where we can analyze space and time taken to run our total code.

Considering only Insertion function :

**Execution Time:** 0.042431687 This is the time taken for executing the insertion function in our code.

Considering only Delete function :

**Execution Time:** 0.00106405 we execute both insertion and deletion together and then subtract the time of insertion so that we get the time required for deletion.

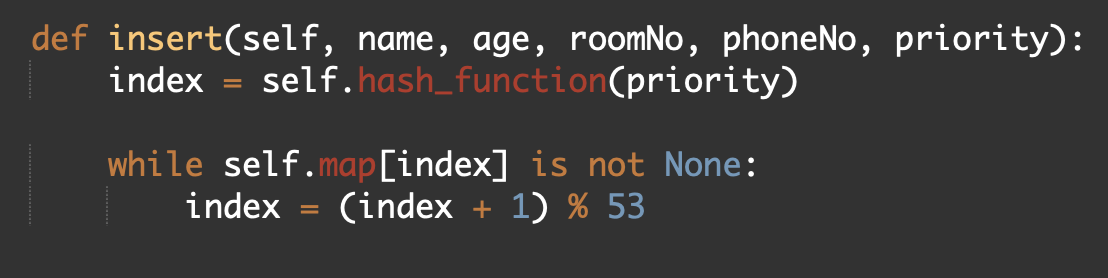
Considering only Search function :

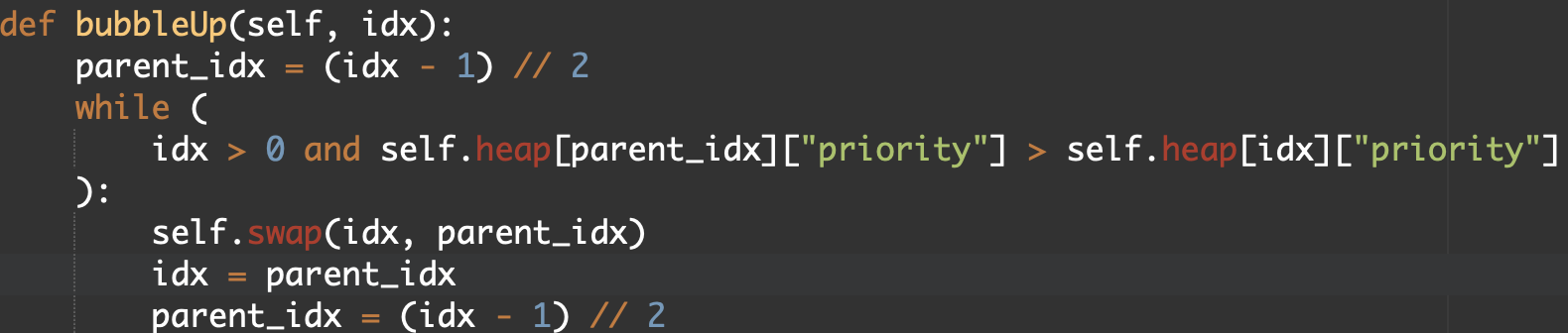
**Execution Time:** 0.015568698 we execute both insertion and search together and then subtract the time of insertion so that we get the time required for search.

Methodology is just executing the code in hpoj and noting down the time taken for required.

Interpreting the No of comparisons taken in hash-map and min-heap for Insertion :

|  |
| --- |
| None |
| None |
| None |
| None |
| None |

 While insertion this is a position there is a comparison in the hash map where it checks whether the position it is pointing to is None or is there any value found in it as in our code there are no collisions there is only one comparison with None.

 In bubble-up function this is heapification while heapification when an element is inserted into array it compares with its parent and if there is any interchange required the interchange process will happen directly again after interchanging there will be another comparison with its parent and again if it is required the process is done based on our code there are only maximum of 2 comparisons required for any number of inputs.

These are results for 25 inputs in random order 2,3,2,2,2,3,2,2,3,2,2,3,3,3,2,2,2,2,2,2,2,2,2,3,2

So maximum number of total comparison for hash map and min heap together are 3.

Also there are 25 data sets used for the experimental evaluation.

Interpreting the No of comparisons taken in hash-map and min-heap for Deletion :

hpq.delete(name\_priority\_dict["Liam"]) -> 15

hpq.delete(name\_priority\_dict["Emily"]) -> 18

hpq.delete(name\_priority\_dict["Benjamin"]) -> 6

hpq.delete(name\_priority\_dict["Noah"]) -> 9

hpq.delete(name\_priority\_dict["Ava"]) ->12

hpq.delete(name\_priority\_dict["Isabella"]) -> 12

hpq.delete(name\_priority\_dict["Sophia2"]) -> 6

hpq.delete(name\_priority\_dict["Olivia"]) -> 12

hpq.delete(name\_priority\_dict["Alexander"]) -> 9 these are some comparisons of done in Min-Heap when an item is got deleted.

Discussion :

The Hybrid Priority Queue, which combines a min heap and a hash map, offers several advantages in real-world scenarios. Let's discuss its practicality, effectiveness, limitations, challenges, and potential improvements.

Practicality and Effectiveness:

1. Efficient Priority Queue: The hybrid data structure provides an efficient priority queue implementation. The min heap ensures that the element with the highest priority can be accessed in constant time, while the hash map enables efficient searching and updating of elements based on their names.
2. Fast Insertion and Deletion: Insertion and deletion operations in the hybrid data structure have a time complexity of O(log n), where n is the number of elements. This makes it practical and efficient for handling dynamic data with changing priorities.
3. Flexibility: The hybrid data structure allows you to prioritize elements based on their priorities while also providing a convenient way to retrieve elements by their names. This flexibility is beneficial in scenarios where you need to manage and manipulate data based on both priority and name.
4. Space Efficiency: The hybrid data structure optimizes space by using a hash map to store element details, avoiding the need to duplicate the elements in the min heap. This can be advantageous when dealing with large datasets.

Limitations and Challenges:

1. Size Limitation: The hybrid data structure has a fixed size determined during initialization. If the number of elements exceeds this size, it may lead to collisions in the hash map or inability to insert new elements. Handling this limitation requires careful selection of the initial size or implementing resizing strategies.
2. Collision Resolution: The hash map used in the hybrid data structure may encounter collisions, resulting in performance degradation. Proper collision resolution techniques, such as chaining or open addressing, need to be employed to mitigate this issue.
3. Memory Overhead: The hybrid data structure requires additional memory to store the hash map alongside the min heap. While this overhead is generally acceptable, it should be considered when working with limited memory resources.

Potential Future Improvements:

1. Dynamic Resizing: Implementing dynamic resizing strategies for the hybrid data structure can allow it to adapt to changing data sizes automatically. This would ensure efficient memory utilization and eliminate the size limitation challenge.
2. Enhanced Hashing: Utilizing advanced hashing techniques, such as cryptographic hash functions or perfect hashing, can improve the hash map's performance and reduce collision probabilities.
3. Additional Operations: Expanding the hybrid data structure with additional operations, such as extracting the minimum element based on name or priority, can further enhance its functionality and usability.
4. Performance Optimization: Analyzing and optimizing the algorithms and data structures used in the hybrid data structure can lead to better overall performance, reducing time complexities or memory consumption where possible.

Overall, the hybrid data structure combining a min heap and a hash map offers practicality and effectiveness in managing priorities and enabling efficient element retrieval by name. While it has limitations and challenges, addressing them through dynamic resizing, enhanced hashing, additional operations, and performance optimization can make it even more valuable in real-world scenarios.

Conclusion :

The project focused on implementing a hybrid data structure, specifically a hybrid priority queue, using a combination of a min heap and a hash map. The outcomes and findings of the project are as follows:

1. Practical Applications: The hybrid priority queue has practical applications in scenarios where elements need to be prioritized based on their priorities and retrieved or updated efficiently using their names. Examples include task scheduling, job prioritization, event handling, and resource allocation, hospital management, Online ticketing.
2. Performance Analysis: The hybrid data structure offers efficient performance in terms of insertion, deletion, and retrieval operations. Insertion and deletion have a time complexity of O(log n), where n is the number of elements. Retrieving an element by name or obtaining the maximum priority element takes constant time, making the data structure suitable for real-time or time-critical applications.
3. Efficiency: The hybrid data structure optimizes space utilization by storing the element details in a hash map, avoiding duplication in the min heap. This results in efficient memory usage, particularly when dealing with large datasets or when memory resources are limited.
4. Overall Success: The project can be considered successful as it achieved its objectives of implementing a hybrid priority queue using a min heap and a hash map. The data structure provides a flexible and efficient solution for managing prioritized elements and enables fast retrieval and update operations based on element names.

Insights Gained:

1. Balance between Priority and Name: The hybrid data structure strikes a balance between prioritizing elements based on their priorities and allowing efficient access to elements using their names. This flexibility provides a practical and versatile solution for various real-world scenarios.
2. Importance of Data Structure Selection: The project highlighted the significance of selecting appropriate data structures based on the requirements of the problem. Combining different data structures can lead to enhanced performance and functionality.
3. Trade-offs and Optimization: The project emphasized the trade-offs between different operations and the need for optimization strategies. For example, the min heap ensures efficient priority-based operations, while the hash map facilitates fast name-based operations. Balancing these operations and optimizing algorithms is crucial for overall efficiency.

In conclusion, the project successfully implemented a hybrid priority queue using a min heap and a hash map, demonstrating its practical applications and efficient performance. The project provided insights into the importance of data structure selection, trade-offs, and optimization strategies. The hybrid data structure offers a valuable solution for managing prioritized elements and accessing them efficiently by name.