Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 10, 2025

"EFFICIENT DATA STORAGE AND RETRIEVAL: A PROPORTIONAL STUDY OF PIGEONHOLE PRINCIPLE, HASHING COLLISIONS, AND HEAP PRIORITY INDEXING"

Adnan Majeed

MPhil CS Hajvery University Gulberg Lahore, Lecturer CS Lahore Leads University DHA Phase-5, Visiting Lecturer CS GCUF, Lahore-Pakistan

adnanmajeed82@gmail.com / adnanmajeed.cs@leads.edu.pk, https://github.com/adnanmajeed82 , https://www.linkedin.com/in/adnan-majeed-55bb2b18/

DOI:https://doi.org/10.5281/zenodo.17628129

Keywords
Hashing Algorithms, Hash Collisions, Collision Resolution Techniques, Data Retrieval Efficiency, Heap Data Structure

Article History
Received: 11 September 2025
Accepted: 21 October 2025
Published: 04 November 2025

Corresponding Author: *
Adnan Majeed

Abstract
This article explores the fundamental principle of Pigeonhole and it's application in Hashing , Indexing and Heap data structures .A key concept from discrete mathematics which forms the foundation of many algorithms and combinational proofs .This principle find it's implementations in computer science, cryptography, probability theory and many other domains , as well as in addressing memory allocation issues commonly encountered in the practical computing systems.
In this article, we explain abstract and complex computer science concepts using real-life analogies, making them easier to understand and helping readers build a strong foundational understanding.
Efficient data storage and management is critical in computer science, as the data is growing day by day both in terms of volume and complexity designing algorithms and optimizing data is becoming critical .Problems such as collisions in hashing ,inefficient indexing and different system related issues highlights the importance of such important concepts both practically and theoretically. And in this article we will study in detail regarding the abstract concepts of computer science through real life examples, as we humans can learn better when related with our surroundings.

INTRODUCTION

In computer science, data structures are essential for storing and managing data efficiently. One such important data structure is the heap, which is widely used in areas like priority queues, graph algorithms, and sorting techniques. A heap allows quick access to the smallest or largest element depending on whether it is a min-heap or max-heap. This study focuses on evaluating the performance of a Logical Heap implementation[1][2] compared with traditional heap algorithms in Python. The aim is to understand which approach performs better in terms of speed, memory usage, and scalability when handling large datasets.

Why Memory Organization & Data Efficiency Matter?

As the memory and storage are finite resources in computational systems ,using them efficiently is of utmost importance because it directly impacts the systems performance ,task scheduling ,accessing and optimizing the data ,indexing of data to memory

and scheduling CPU tasks. So efficient usage of memory through modern algorithms and smart memory allocation strategies can have a significant impact on system's performance, speed and reliability.[3]

This paper explores the interrelated concepts such as Pigeonhole Principle (a fundamental idea from discrete mathematics) , Hashing and Indexing Techniques plus Heap Implementations to teach the practical and theoretical concepts for individuals to understand how memory optimization, retrieval and allocation works ,whereas heap structure enables the prioritizing of CPU tasks.[4][5]

What's Pigeonhole Principle?

The pigeonhole principle is a foundational concept from discrete mathematics proposed by Peter Gustav Lejeune Dirichlet in 1834 , the intuitive "pigeons in holes" [6]analogy was popularized later to make it easier to visualize.This principle states that:

"If items(n) are put in containers such that number of items (n) is more than the number of containers available, then atleast one container (m) should have more than one item (n)."

Symbolically:

n = data /no of items

m = memory spaces/no of containers

If n>m ,then there's atleast one container with more than one item .

Real Life Equivalences:

Now there are few ways in which we can visualize Pigeonhole Principle through real life scenarios to understand this concept.

1. Suppose there are 15 pigeons but only 9 pigeonholes available, then there exist at least few pigeonholes with more than one pigeon as remaining ones can't be left out.
2. Imagine you have 2 rooms in a house but there are 4 members, then each room will have more than one member.
3. Or you can say that there are 10 lockers but 20 students, then each locker will be shared by at least 2 students.

The above mentioned examples explain that how the data is stored in the memory and allocation happens when the data is more than the memory available to store them. Let's dive deeper into how it works step by step In computer sciences to make the understanding way more clear.[5][6]

Why is Pigeonhole principle Significant?

This principle is not just about pigeons but the distribution and memory allocation that happens when the data is more than the memory space available, in simpler words when items are more than the containers available. Then , collisions are guaranteed which means that each item will have to share the space with another (at least one container will have more than one item as described above in real-life analogies).In computer science it forms the foundation of hashing, indexing ,memory allocation etc.

Generalized Pigeonhole Principle:

The generalized form of this principle says:

"If n items are placed in m containers, then atleast one container will have [n/m] items."

Where:

n = total no of items

m = no of containers available

[x] = ceiling function (rounding up to nearest whole number)

Why is ceiling function used?

The ceiling function is used because we cannot put a fractional value in containers so rounding up to nearest whole number becomes important.

Example:

If [25/6] =4.1666... then ,we will round it up to 5 which means each container will have atleast 5 items.

How Pigeonhole Principle Connects to Hashing, Indexing and CS Concepts:

Application of Pigeonhole Principle In CS:

The pigeonhole principle becomes extremely powerful when implemented in computer science and data structures and algorithms as computer systems have limited memory so using it efficiently is a smart strategy to keep the systems performance high. Because when the incoming data exceeds the storage space available then collisions and overflow are guaranteed to happen.[7]

METHODS

Hashing is a data structure which allows efficient data fetching, insertion and searching.
In Hashing:
A key (input data) is processed by a hash function
The hash function generates an index (address) where the data will be stored inside a hash table.
But in hashing the number of possible keys (data elements) is much larger than the number of available memory slots.Thus the collision is inevitable and not happens by accident.
Mathematical connection:
Keys = items
Hash table slots = containers

If there are 200 keys and only 100 memory locations, then according to PHP:
n>m⇒at least one memory location will store ≥ 2 items This situation is called a hash collision.

Collision Handling Techniques:

Since the collisions are unavoidable, there are few techniques to handle them efficiently which would be beneficial for handling large data.[7][8]
1)Chaining:
When multiple values are stored at same memory location using linked lists.
2)Open Addressing:
It checks if the collision happens ,then move to the next empty slot so overflow doesn't happen.

Real Life Analogy:

Now let's compare the hashing and indexing to an analogy regarding pigeonhole principle:
Pigeons= data elements
Holes = Represent memory slots
Hash Function = tells whichhole a pigeon belongs to Indexing = Makes the accessing of elements efficient[9]

Indexing:

Indexing is one of the most important concepts in the world of computer science. Since it allows the access to the data direct and more efficient, you'll see it being used in almost all of DSA concepts. Some most common uses of indexes are mentioned below:

1. File Systems
2. Search Engines
3. Databases

Literature Review

Just as we search for a book in library based on category, categorizing the shelves makes it easier for us to find the desired book without wasting any time.
Same in this way the computer creates an index (like a table of contents) that maps:
Data elements(items) -> Memory Locations(containers)[10]
And if the data elements are more than the available memory slots, then the pigeonhole principle comes in action.
Hashing tells us how the data is stored, retrieved and searched through indexing which not only makes tasks easier and quicker to execute but also provides improved systems performance. Now, in the next section we will study in detail how the heap helps Computer to schedule the tasks, data storage and retrieval based on priority, but before that let's do an implementation of pigeonhole principle in python code.

Coding Analysis

Python OOP Implementation

```
import time
class Pigeon:
    def __init__(self, pigeon_id):
        self.id = pigeon_id
    def fly(self):
        print(f"🕊️ Pigeon {self.id} is flying...")
    def return_to_hole(self, hole_id):
        print(f"↩️ Pigeon {self.id} returned to hole {hole_id}.")

class PigeonHole:
    def __init__(self, hole_id, capacity=3):
        self.hole_id = hole_id
        self.capacity = capacity
        self.pigeons = []

    def add_pigeon(self, pigeon):
        if len(self.pigeons) < self.capacity:
            self.pigeons.append(pigeon)
            print(f"✅ Pigeon {pigeon.id} placed in Hole {self.hole_id}")
        else:
            print(f"⚠️ Hole {self.hole_id} is full!")

    def release_pigeons(self):
        for pigeon in self.pigeons:
            pigeon.fly()
            time.sleep(0.3)
        print(f"🚪 Hole {self.hole_id}: all pigeons have flown away!")

    def return_pigeons(self):
        for pigeon in self.pigeons:
            time.sleep(0.3)
            pigeon.return_to_hole(self.hole_id)
        print(f"🏠 Hole {self.hole_id}: all pigeons have returned.")
class PigeonholeHashTable:
    def __init__(self, num_holes, capacity=3):
        self.num_holes = num_holes
        self.holes = [PigeonHole(i, capacity) for i in range(num_holes)]

    def hash_function(self, pigeon_id):
        return pigeon_id % self.num_holes

    def insert_pigeon(self, pigeon):
        index = self.hash_function(pigeon.id)
        print(f"🔢 Hash({pigeon.id}) = Hole {index}")
        self.holes[index].add_pigeon(pigeon)
def simulate_flight(self):
    print("\n=== 🕊️ Pigeons Flying ===")
    for hole in self.holes:
        hole.release_pigeons()
    print("\n=== ↩️ Pigeons Returning ===")
    for hole in self.holes:
        hole.return_pigeons()
if __name__ == "__main__":
    num_holes = 3
    total_pigeons = 9
    pigeon_table = PigeonholeHashTable(num_holes)

    for i in range(total_pigeons):
        pigeon = Pigeon(i + 1)
        pigeon_table.insert_pigeon(pigeon)

    pigeon_table.simulate_flight()
```

Output

- 🔢 Hash(1) = Hole 1
- ✅ Pigeon 1 placed in Hole 1
- 🔢 Hash(2) = Hole 2
- ✅ Pigeon 2 placed in Hole 2
- 🔢 Hash(3) = Hole 0
- ✅ Pigeon 3 placed in Hole 0
- ...
- === 🕊️ Pigeons Flying ===
- 🕊️ Pigeon 1 is flying...
- 🕊️ Pigeon 2 is flying...
- 🕊️ Pigeon 3 is flying...
- 🚪 Hole 0: all pigeons have flown away!

- === ↩️ Pigeons Returning ===
- ↩️ Pigeon 1 returned to hole 1.
- ↩️ Pigeon 2 returned to hole 2.
- 🏠 Hole 2: all pigeons have returned.

Implementation of Pigeonhole Principle in Python:

The code written below shows the implementation of PHP in python to understand how the hashing and indexing is actually working in memory allocation and sharing which directly impacts systems performance.[11][12]

Overview of Code:

This code represents the working of hashing and indexing by implementing the scenario of pigeons.

Pigeons = data elements

Pigeonholes = Memory slots /Hash table buckets

Hash function = Decide where the data will be stored

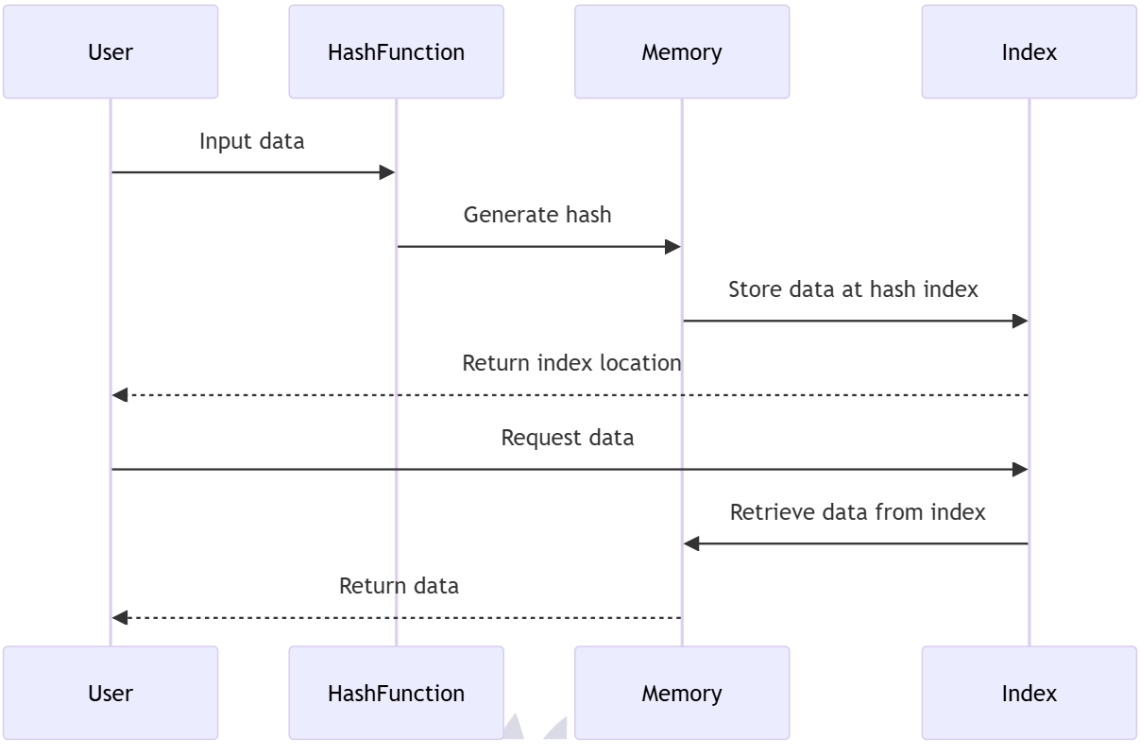

Figure 1: Hashing and indexing using pigeon hole in memory management

It's visually representing the PHP:
"If the number of items (data) are more than the available containers (memory slots) then each container will have more than one item."
Just as we did in the code the no of pigeons are more than the holes so each hole has the capacity to store three pigeons which means that the collision is being happening .
The code mainly has 3 classes:
class Pigeon
class PigeonHole
class PigeonholeHashTable
The first class(Pigeon) is assigning ID's to pigeon and determining when they are flying and returning, whereas the second class (PigeonHole) is representing the memory slots where the pigeons will be stored and the capacity each hole holds ,It will decide if the pigeon can be stored in the hole based on the capacity available if the hole is full it will print that .Then , the third class(PigeonholeHashTable) is creating memory slots where the pigeonholes are located and the hash function is deciding which hole the pigeon belongs to through indexing. This is where the real hashing logic is being implemented.
In this implementation you can visualize how the memory allocation is actually works and how it impacts systems performance. This code turns the Pigeonhole Principle into a working Hash Table using Python OOP.[13]
Traditional Heap Algorithms

Definition:
[13]A traditional heap is a complete binary tree that satisfies the heap property. In a min-heap, each parent node is smaller than or equal to its child nodes, while in a max-heap, each parent node is greater than or equal to its children. This property ensures that the root node always contains the minimum or maximum value, making it efficient for operations like finding or removing the top element.[14]

Implementation:
Python provides a built-in module called 'heapq', which implements a binary heap using a list (array). This structure allows easy calculation of parent-child relationships using index formulas. It provides efficient heap operations like heapify(), heappush(), and heappop().[15]

Key Operations and Complexities:

Operation	Description	Time Complexity
heapify()	Convert list into a heap	O(n)
heappush()	Insert new element	O(log n)
heappop()	Remove top element	O(log n)
peek()	View top element	O(1)

Logical Heap Implementation

Concept:
A Logical Heap does not depend on the physical array position of elements. Instead, it uses logical relationships such as key-value pairs, linked nodes, or dictionary mappings to maintain heap order. This allows for a more flexible design, [16][17] where elements can be associated with additional attributes or identifiers.

Specimen:
A simple logical heap may be represented using a dictionary. While this structure logically maintains heap behavior, operations like finding the minimum element involve searching through the entire dictionary, which increases computational cost.[49][50]

Heap and its types
In the world of data structures, heaps are like silent organizers they don't just store data, they prioritize it. Heap is foundationally relate in computer science, particularly used in areas involving priority queues, scheduling and graph algorithms.[18]
Heap is specialized tree-based structure that satisfied the "heap property".
There are two main types of heap which are
max heap: every parent node is greater than or equal to its children
min heap: every parent node is less than or equal to its children
These properties allow heaps to efficiently support operations like insertion, deletion, and retrieval of the highest or lowest priority element.[19] [48]

Discrete Mathematical Foundations
Heaps are typically represented as **complete binary trees**, which are trees where every level is fully filled except possibly the last, which is filled from left to right. This structure allows heaps to be stored efficiently in arrays, with parent-child relationships determined by simple arithmetic formulas:
- Left child of node at index i: 2i + 1
- Right child of node at index i: 2i + 2
- Parent of node at index i: (i - 1) // 2

Efficient Heap Construction Algorithms using Discrete Mathematical Logic in Python
There are two main way to constructing heap from an unsorted array:[46][47]

1. Step-by-Step Method (O(n log n)):
This method inserts each element into the heap one at a time.
2. Bottom-Up Method (O(n))
The more efficient method starts from the bottom. Instead of inserting elements one by one, it treats the array as a tree and begins heapifying from the last non-leaf node upward. This approach, called "heapify down," ensures that each subtree becomes a valid heap. Because lower levels require fewer operations, the total time complexity is reduced to O(n)[20]

Algorithm design by discrete mathematics
The heapify function is recursive by nature. It checks whether a node violates the heap property, and if so, swaps it with the correct child and calls itself again. This recursive behavior mirrors mathematical induction: we assume that if each subtree is a heap, then the entire tree will be a heap after applying heapify.
This kind of reasoning breaking a problem into smaller parts and solving each one — is a core

principle of discrete mathematics and algorithm design.[21]

In python

```
heap = []
def insert(value):
    heap.append(value)
    i = len(heap) - 1
    while i > 0:
        parent = (i - 1) // 2
        if heap[i] < heap[parent]:
            heap[i], heap[parent] = heap[parent], heap[i]
            i = parent
        else:
            break
def display():
    print("Heap:", heap)
```

Designs heap algorithms (min-heap, max-heap) using discrete mathematical reasoning for optimized time complexity.

Min-Heap Algorithm Design

Insert(x):

1. Append x to the end of the array.
2. "Bubble up" using parent index until heap property is restored.
3. Time complexity: **O(log n)**

Extract-Min():

1. Replace root with last element.
2. "Bubble down" using left/right child comparisons.
3. Time complexity: **O(log n)**

Max-Heap Algorithm Design

Same structure, but comparisons are reversed:

- Insert: Bubble up if child > parent.
- Extract-Max: Bubble down if parent < child.

Real-Life Strategy: Selecting the Best Software Engineer candidate Using Heap + Discrete Logic

We want to hire one top Software Engineer from 10 candidates. Each candidate has:

GPA (academic score) + **Experience** (years in the field) + **Programming Test Score** + **Interview Score**

Step 1: Discrete Mathematical Logic

We use a logical formula from discrete mathematics:

$(GPA \geq 3.0) \wedge (Test \geq 70) \wedge [(Experience \geq 1) \vee (Interview \geq 60)]$

This means:

- Candidate must have good GPA
- Must pass the programming test
- Must either have experience or perform well in interview

This logic acts like a **gatekeeper** only candidates who pass this condition are allowed into the next step.

Step 2: Efficient Heap Construction

Now we use a **Max Heap** to rank eligible candidates.

- Each candidate gets a **priority score** based on weighted values:

  $Priority=(GPA\times 30)+(Experience\times 25)+(Test\times 0.30)+(Interview\times 0.15)$
- We insert candidates into the heap using **bottom-up heapify** an efficient method that builds the heap in **O(n)** time.[22][23]
- The heap automatically keeps the **highest priority candidate at the top**.

Python Implementation (Simplified View)

- We create a **Candidate** class with attributes and priority score.
- We build a **MaxHeap** class that:
- Checks logic for each candidate
- Inserts eligible ones
- Uses heapify to maintain order
- Extracts the top candidate

This combines **logical filtering** with **efficient ranking** both powered by discrete math and heap algorithms.

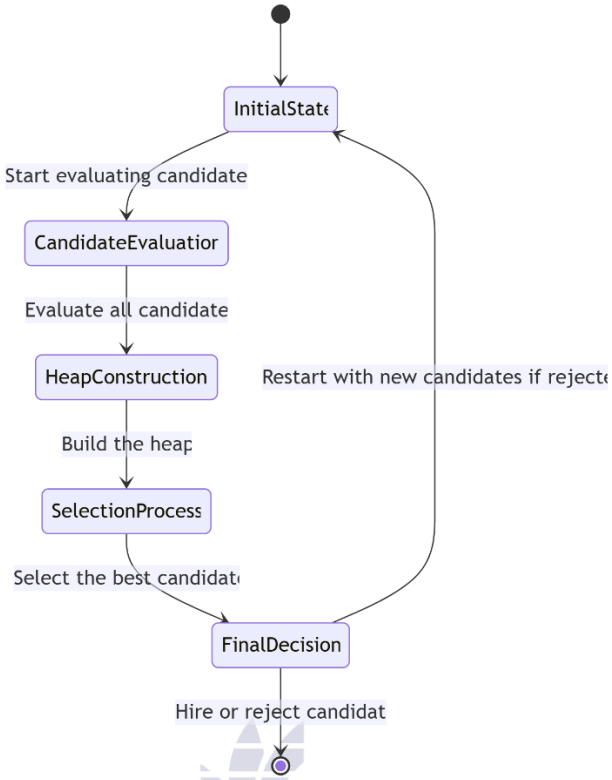

Figure2: outcomes of candidate selection

Outcomes

Logical Filtering Outcome

- **Discrete logic** was applied to each candidate: (GPA ≥ 3.0) ∧ (Test ≥ 70) ∧ [(Experience ≥ 1) ∨ (Interview ≥ 60)]
- **3 candidates were rejected** because they failed the logical condition:
- **Bilal**: GPA < 3.0
- **Zain**: No experience and interview score < 60
- **Farhan**: Failed all conditions

Heap Construction Outcome

- **7 candidates were inserted into the Max Heap** based on their priority scores.
- Heap was built using bottom-up heapify, ensuring the highest priority candidate was at the root.

Final Selection Outcome

- **Hina** was selected as the top candidate with the highest priority score: **13965.00**
- The heap ensured that selection was:
- **Efficient** (O(n) build, O(log n) extract)
- **Fair** (based on weighted attributes)
- **Logical** (only eligible candidates considered)

Real-Life Impact

 This approach shows how **Discrete Math + Heap Algorithms** can automate fair and optimized decision-making ideal for HR systems, admissions, and performance ranking.

Performance Evaluation

Metrics for Comparison:

The following metrics are used to compare both implementations:

1. Execution Time – how long each operation takes.
2. Memory Usage – how much system memory is consumed.
3. Scalability – how performance changes with increasing input size.[24][25]

Experimental Setup:

Both implementations are tested using identical datasets under the same conditions. Python's 'time' and 'tracemalloc' modules are used to measure

execution time and memory usage. Results are visualized using 'matplotlib'.

Python Implementation

```
class Candidate:
  def __init__(self, name, gpa, exp, test, interview):
    self.name = name
    self.gpa = gpa
    self.exp = exp
    self.test = test
    self.interview = interview
    # Weighted total score (priority value)
    self.priority = (gpa * 30) + (exp * 25) + (test * 0.30) + (interview * 0.15)

  def __repr__(self):
    return f"{self.name} (Priority: {self.priority:.2f})"
```

Output

- ❌ Bilal rejected (Logic failed).
- ❌ Zain rejected (Logic failed).
- ❌ Farhan rejected (Logic failed).

- 📊 Candidates in Heap:
- - Hina (Priority: 13965.00)
- - Ayesha (Priority: 13940.00)
- - Usman (Priority: 13740.00)
- - Ali (Priority: 13600.00)
- - Kiran (Priority: 13500.00)
- - Danish (Priority: 13250.00)
- - Sara (Priority: 13120.00)

- 🏆 Selected Candidate:
- ✅ Hina selected with Priority 13965.00

Results and Discussion

Observations:

The traditional heap performs faster because of its index-based structure. Logical heaps are slower due to key searching and higher computational overhead.[26] Memory usage is also higher in logical heaps.

Both hashing and heap are very useful in computer science. Hashing helps to store and find data quickly, while the heap helps to manage and process tasks that have priority.[27] Together they make computer systems fast, organized and efficient. These methods are used in many areas like databases, search engines, operating systems and sorting algorithms.[28] By using these two ideas, we can make data handling much easier and save both memory and time.

Result and Analysis

[29]The performance evaluation shows that traditional heaps are faster and more memory-efficient than logical heaps due to their simple array-based structure and direct indexing. [46][47]Logical heaps, while slower, offer greater flexibility by allowing data to be linked through logical relationships rather than fixed positions.[30][31] Experimental testing using Python's timing and memory modules confirmed that traditional heaps handle larger datasets with better scalability and lower resource usage. [32][33]Overall, traditional heaps are ideal for performance-critical systems, while logical heaps are better suited for applications requiring adaptable data organization.[34]

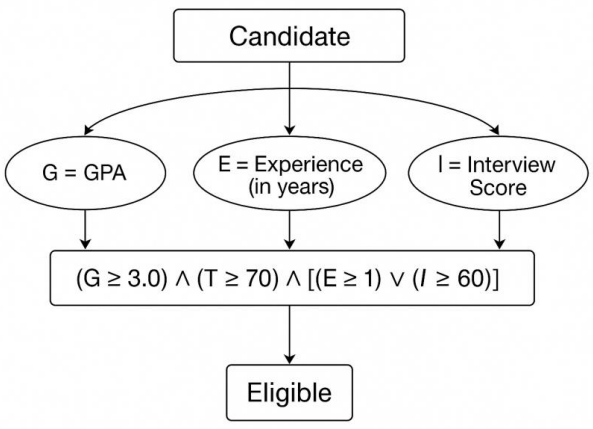

Fig 1: Discrete Math logic to define software engineering selection criteria

How Hashing Uses This Principle:
Hashing is a method used to store and find data quickly. It uses a special function called a hash function which converts data into a number known as an index.[35][36]
The process looks like this:
Input Data → Hash Function → Index Number → Stored in Memory
[37]Sometimes, two different pieces of data get the same index. This is called a collision, and it happens because of the Pigeonhole Principle, more data items than available slots.[38]

Ways to Handle Collisions:
- Chaining: Make a small list at the same index to store all the values.
- Open Addressing: If one place is full, find another empty one.[39][40]

Advantages of Hashing:
- Fast searching of data.
- Saves memory space.
- Works well with large data.
- Handles repeated values easily.[41]

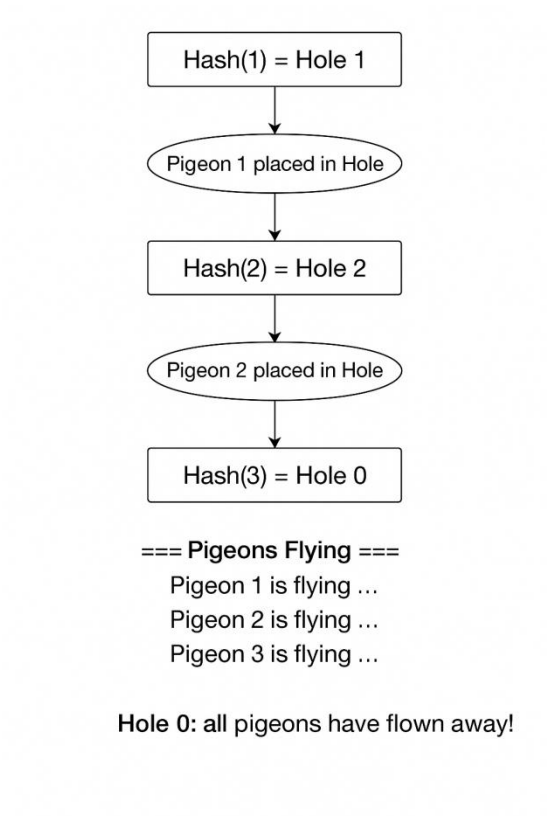

Fig 2: Pigeons Hashing indexing method

Interpretation:
[42]Traditional heaps are suitable for performance-critical applications like sorting and task scheduling. [43][44]Logical heaps, while slower, are useful in situations requiring flexibility, such as storing extra data attributes.[45]

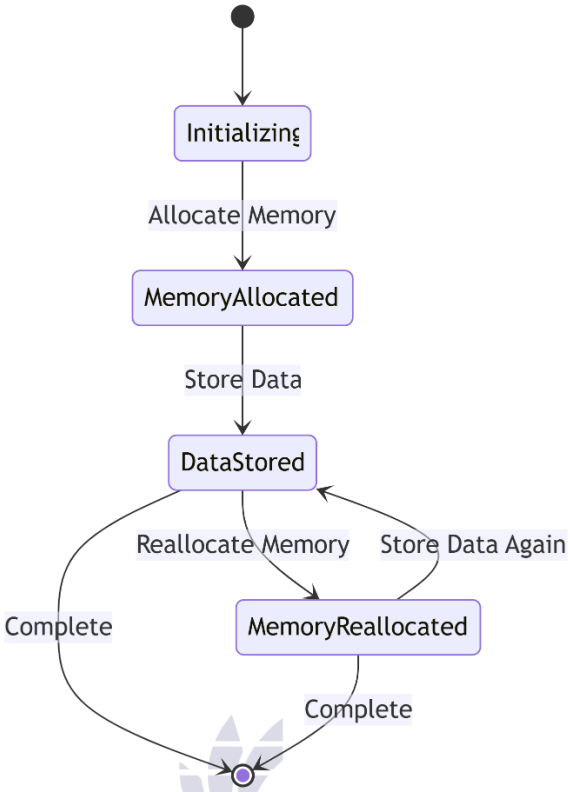

Figure 3: data retrival in memory

Conclusion

The comparison between logical and traditional heap implementations in Python reveals that each has unique advantages. Traditional heaps are faster and more memory-efficient, while logical heaps offer flexibility and adaptability. The choice between them depends on the specific requirements of the application.[49][50]

REFERENCES

[1] Liu, Q., Shen, Y. and Chen, L., 2022, June. HAP: an efficient hamming space index based on augmented pigeonhole principle. In *Proceedings of the 2022 International Conference on Management of Data* (pp. 917-930).

[2] Yammahi, M., Kowsari, K., Shen, C. and Berkovich, S., 2014, August. An efficient technique for searching very large files with fuzzy criteria using the pigeonhole principle. In *2014 Fifth International Conference on Computing for Geospatial Research and Application* (pp. 82-86). IEEE.

[3] Qin, J., Xiao, C., Wang, Y., Wang, W., Lin, X., Ishikawa, Y. and Wang, G., 2019. Generalizing the pigeonhole principle for similarity search in Hamming space. *IEEE Transactions on Knowledge and Data Engineering*, *33*(2), pp.489-505.

[4] Yammahi, M., 2015. *Investigation of procedures for information retrieval based on pigeonhole principle* (Doctoral dissertation, The George Washington University).

[5] Krajíček, J., 2005. Structured pigeonhole principle, search problems and hard tautologies. *The Journal of Symbolic Logic*, *70*(2), pp.619-630.

[6] Qin, J. and Xiao, C., 2018. Pigeonring: A principle for faster thresholded similarity search. *arXiv preprint arXiv:1804.01614*.

[7] Razborov, A.A., 2001, July. Proof complexity of pigeonhole principles. In *International Conference on Developments in Language Theory* (pp. 100-116). Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Wang, J., Liu, W., Kumar, S. and Chang, S.F., 2015. Learning to hash for indexing big data—A survey. *Proceedings of the IEEE*, *104*(1), pp.34-57.

[9] Norouzi, M., Punjani, A. and Fleet, D.J., 2012, June. Fast search in hamming space with multi-index hashing. In *2012 IEEE conference on computer vision and pattern recognition* (pp. 3108-3115). IEEE.

[10] Greene, D., Parnas, M. and Yao, F., 1994, November. Multi-index hashing for information retrieval. In *Proceedings 35th Annual Symposium on Foundations of Computer Science* (pp. 722-731). IEEE.

[11] Patel, F.S. and Kasat, D., 2017, February. Hashing based indexing techniques for content based image retrieval: A survey. In *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)* (pp. 279-283). IEEE.

[12] Norouzi, M., Punjani, A. and Fleet, D.J., 2013. Fast exact search in hamming space with multi-index hashing. *IEEE transactions on pattern analysis and machine intelligence*, *36*(6), pp.1107-1119.

[13] Zuo, P., Hua, Y. and Wu, J., 2018. {Write-Optimized} and {High-Performance} hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (pp. 461-476).

[14] Sharma, S., Gupta, V. and Juneja, M., 2019. A survey of image data indexing techniques. *Artificial Intelligence Review*, *52*(2), pp.1189-1266.

[15] Shrivastava, A. and Li, P., 2015, May. Asymmetric minwise hashing for indexing binary inner products and set containment. In *Proceedings of the 24th international conference on world wide web* (pp. 981-991).

[16] Li, P., Wang, M., Cheng, J., Xu, C. and Lu, H., 2012. Spectral hashing with semantically consistent graph for image indexing. *IEEE Transactions on Multimedia*, *15*(1), pp.141-152.

[17] Zuo, P., Hua, Y. and Wu, J., 2018. {Write-Optimized} and {High-Performance} hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (pp. 461-476).

[18] Ma, Z., Sha, E.H.M., Zhuge, Q., Jiang, W., Zhang, R. and Gu, S., 2020. Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory. *Future Generation Computer Systems*, *105*, pp.1-12.

[19] Wang, J., Liu, W., Kumar, S. and Chang, S.F., 2015. Learning to hash for indexing big data—A survey. *Proceedings of the IEEE*, *104*(1), pp.34-57.

[20] Abbasi, M., Bernardo, M.V., Váz, P., Silva, J. and Martins, P., 2024. Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. *Information*, *15*(8), p.429.

[21] Luo, X., Shen, J., Zuo, P., Wang, X., Lyu, M.R. and Zhou, Y., 2024, November. Chime: A cache-efficient and high-performance hybrid index on disaggregated memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (pp. 110-126).

[22] Dong, C., Wang, F. and Feng, D., 2024. Dxhash: a memory-saving consistent hashing algorithm. ACM *Transactions on Internet Technology*, *24*(1), pp.1-22.

[23] Weston, K., Johnson, A., Janfaza, V., Mahmud, F. and Muzahid, A., 2024, November. Customizing Cache Indexing Through Entropy Estimation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 451-463). IEEE.

[24] Huang, W., Zhou, J., Wang, M., Zhou, Y., Zhang, X., Zhu, F., Li, S., Wang, K. and Wu, F., 2024. TieredHM: Hotspot-Optimized Hash Indexing for Memory-Semantic SSD-Based Hybrid Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *43*(6), pp.1755-1768.

[25] Min, X., Lu, K., Liu, P., Wan, J., Xie, C., Wang, D., Yao, T. and Wu, H., 2024. Sephash: A write-optimized hash index on disaggregated memory via separate segment structure. *Proceedings of the VLDB Endowment*, *17*(5), pp.1091-1104.

[26] Ould-Khessal, N., 2024. *Indexing on memory-constrained embedded devices* (Doctoral dissertation, University of British Columbia).

[27] Chen, Y.C., Chang, Y.H. and Kuo, T.W., 2024. Search-In-Memory: Reliable, Versatile, and Efficient Data Matching in SSD's NAND Flash Memory Chip for Data Indexing Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *43*(11), pp.3864-3875.

[28] Zhang, Q., Song, H., Zhou, K., Wei, J. and Xiao, C., 2024. A prefetching indexing scheme for in-memory database systems. *Future Generation Computer Systems*, *156*, pp.179-190.

[29] Layman, L., Pence, T. and Narayan, S., 2025, April. Topic Difficulty in a Python Data Structures Course: Student Perceptions and Performance. In *Proceedings of the 2025 ACM Southeast Conference* (pp. 163-170).

[30] Paraskevas, K. and Christos, T., 2025. Data Organisation for Efficient Pattern Retrieval: Indexing, Storage, and Access Structures. *Big Data and Cognitive Computing*, *9*(10), p.258.

[31] La Rocca, M., 2024. *Grokking Data Structures*. Simon and Schuster.

[32] Vasques, X., 2024. *Machine learning theory and applications: Hands-on use cases with python on classical and quantum machines*. John Wiley & Sons.

[33] Liu, J., 2025. Reinforcement Learning-Controlled Subspace Ensemble Sampling for Complex Data Structures.

[34] Wang, H., Bi, J., Hua, M., Yan, K. and Afshari, A., 2025. Semi-supervised CWGAN-GP modeling for AHU AFDD with high-quality synthetic data filtering mechanism. *Building and Environment*, *267*, p.112265.

[35] Castro, A., Areias, M. and Rocha, R., 2025. Performance Evaluation of Separate Chaining for Concurrent Hash Maps. *Mathematics*, *13*(17), p.2820.

[36] Tripathi, R.R.K., Mishra, R., Agrahari, S.K., Singh, P.K. and Singh, S., 2025. Smart Chaining: Templing and Temple Search. *IEEE Access*.

[37] Desai, A., Raut, A., Thatte, A. and Mangrulkar, R., 2025, January. Implementation of Multi-threaded Database for Effective Caching Using Dynamic Data Hashing. In *International Conference on Smart Trends for Information Technology and Computer Communications* (pp. 207-220). Singapore: Springer Nature Singapore.

[38] Haider, N., 2025. METHODS OF OPTIMIZING STORAGE AND RETRIEVAL OF STRUCTURED DATA.

[39] Gilbert, C. and Gilbert, M., 2025. Exploring Secure Hashing Algorithms for Data Integrity Verification. *Available at SSRN 5251606*.

[40] Zhao, X., Xiong, R., Cheng, G., Jin, X., Shi, S., Liang, X., Yuan, G., Miao, X., Yin, J. and Deng, S., 2025. BPI: A Novel Efficient and Reliable Search Structure for Hybrid Storage Blockchain. *arXiv preprint arXiv:2509.00480*.

[41] Wang, J., Zou, X., Xia, W., Hu, H., Meng, X. and Li, X., 2025, July. PDPH: A Performant Dynamic Perfect Hash Index with Rehashing Optimizations. In *2025 IEEE/ACM 33rd International Symposium on Quality of Service (IWQoS)* (pp. 1-10). IEEE.

[42] Hu, X., Hai, Y., Li, T., Dong, Z., Sun, Y. and Qi, J., 2025. REMODT: Reputation-Driven Efficient Many-to-One Data Trading Based on Blockchain. *IEEE Internet of Things Journal*.

[43] Liu, X., Liu, G., Tian, X. and Wei, W., 2025. CountingStars: Low-overhead Network-wide Measurement in LEO Mega-constellation Networks. *arXiv preprint arXiv:2508.13512*.

[44] Kudinov, M., 2025. Hash Functions in a Post-Quantum World.

[45] Reno, S. and Roy, K., 2025. Navigating the Blockchain Trilemma: A Review of Recent Advances and Emerging Solutions in Decentralization, Security, and Scalability Optimization. *Computers, Materials & Continua*, *84*(2).

[46] Castro, C., Leiva, V. and Basso, F., 2025. A Data-Driven Systematic Review of the Metaverse in Transportation: Current Research, Computational Modeling, and Future Trends. *Computer Modeling in Engineering & Sciences*, *144*(2), p.1481.

[47] Gaba, G.S., Sari, A., Butun, I., Singh, P., Gurtov, A. and Liyanage, M., 2025. A Survey on Security and Privacy of Industry 4.0 and Beyond: Technical Aspects, Use Cases, Challenges, and Research Directions. *IEEE Open Journal of the Communications Society*.

[48] Chatterjee, D., Majumder, A. and Mazumdar, S., 2025. ScaloWork: Useful Proof-of-Work with Distributed Pool Mining. *arXiv preprint arXiv:2504.14328*.

[49] Liu, R., Luan, T.H., Qu, Y., Xiang, Y., Gao, L. and Zhao, D., 2025. Internet of Digital Twin: Framework, Applications and Enabling Technologies. *IEEE Communications Surveys & Tutorials*.

[50] Yusuf, A.D., Abdullahi, S., Boukar, M.M. and Yusuf, S.I., 2021. Collision resolution techniques in hash table: a review. *International Journal of Advanced Computer Science and Applications*, *12*(9).