Efficient Data Retrieval using Searching Algorithms and Queue Management

\*\*Submitted by:\*\*  
AARUSH C S  
ATHUL K KOSHY  
MOHAMMED RIZWAN PP  
MOIDEEN NIHAL

# 1. Introduction

Searching and data management are key operations in software applications dealing with large data. This project explores the implementation and comparison of Linear Search and Binary Search algorithms for efficient data retrieval. It also integrates a Queue data structure to handle incoming data requests in a First-In-First-Out (FIFO) manner, simulating real-world scenarios like customer support ticket handling, CPU scheduling, or printer queue management.

# 2. Problem Statement

In real-time systems, it's essential to search data quickly and manage requests in order. However, linear search is inefficient for large data sets, and binary search requires sorted data. Similarly, managing multiple incoming requests needs a structured system to avoid chaos. This project combines searching and queuing techniques to build an efficient solution.

# 3. Objectives

- Implement Linear Search and Binary Search for data lookup.  
- Compare their efficiency and performance.  
- Design a Queue system to manage user data or tasks.  
- Combine search operations with queue for efficient processing.  
- Provide a simple, intuitive interface for demonstration.

# 4. Literature Review

- Linear Search checks each element one by one; it’s simple but slow (O(n)).  
- Binary Search divides the dataset to search faster (O(log n)) but works only on sorted data.  
- Queue operates in FIFO manner, suitable for scheduling and handling tasks in order.  
- Real-world applications like ticket booking systems, OS job schedulers, and customer service bots use these principles extensively.

# 5. Methodology

- Build two functions for linear and binary search.  
- Create a queue class for enqueue, dequeue, peek, and isEmpty operations.  
- Store user data or tasks in the queue.  
- Implement search operations on the queue data.  
- Track performance using operation counts or time.

# 6. Implementation Tools

- Programming Language: Python  
- Data Structures: List (for array), Queue (custom class or collections module)

# 7. Expected Results and Analysis

- Binary Search should be faster than Linear Search on sorted data.  
- Queue ensures order of task execution.  
- Combined system will allow task addition, retrieval, and search efficiently.

# 8. Challenges and Solutions

| Challenge | Solution |  
|----------|----------|  
| Binary search requires sorted data | Sort the queue data before applying binary search |  
| Queue may become full in static implementation | Use dynamic list-based implementation |  
| Handling real-time tasks with search | Maintain task IDs for search efficiency |

# 9. Conclusion

This project demonstrates the power of combining fundamental algorithms and data structures to build practical systems. With effective implementation of searching algorithms and queue, the system can handle and retrieve data efficiently, showcasing real-time problem-solving skills in data structure applications.

# 10. References

1. “Data Structures and Algorithms in Python” by Goodrich et al.  
2. “Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein  
3. GeeksforGeeks and W3Schools for algorithmic examples and guides.  
4. Official Python Documentation (docs.python.org)

CODE:-

class Queue:

def \_\_init\_\_(self):

self.queue = []

def enqueue(self, value):

self.queue.append(value)

print(f"Enqueued: {value}")

def dequeue(self):

if not self.is\_empty():

removed = self.queue.pop(0)

print(f"Dequeued: {removed}")

return removed

else:

print("Queue is empty.")

return None

def is\_empty(self):

return len(self.queue) == 0

def peek(self):

return self.queue[0] if not self.is\_empty() else None

def display(self):

print("Queue:", self.queue)

def get\_all(self):

return self.queue.copy()

# Linear Search

def linear\_search(arr, target):

for i, value in enumerate(arr):

if value == target:

return i

return -1

# Binary Search (requires sorted list)

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1

# Example Usage

q = Queue()

q.enqueue(15)

q.enqueue(25)

q.enqueue(5)

q.enqueue(30)

q.enqueue(10)

q.display()

# Linear Search

print("\n--- Linear Search ---")

data = q.get\_all()

target = 30

index = linear\_search(data, target)

print(f"Linear Search: {target} found at index {index}" if index != -1 else f"{target} not found.")

# Binary Search (on sorted data)

print("\n--- Binary Search ---")

sorted\_data = sorted(data)

print("Sorted Data for Binary Search:", sorted\_data)

target = 25

index = binary\_search(sorted\_data, target)

print(f"Binary Search: {target} found at index {index}" if index != -1 else f"{target} not found.")