PERFORMANCE REPORT: SCHOOL MANAGEMENT SYSTEM

1. **Time Complexity Analysis**

Module 1: Student Registry (Hash Table)

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| register\_student() | O(1) average | Direct hash-based insertion |
| search\_student() | O(1) average | Direct hash-based lookup |
| update\_student() | O(1) average | Direct hash-based access |
| delete\_student() | O(1) average | Direct hash-based deletion |
| display\_all\_students() | O(n) | Must traverse all elements |

Worst Case Scenario: O(n) when many hash collisions occur, although this is rare with good hash functions.

Module 2: Course Scheduling (Queue)

|  |  |  |
| --- | --- | --- |
| Operarion | Time complexity | Explanantion |
| enrol\_student\_request() | O(1) | appending to end of list |
| process\_queue() | O(n) | processes all n requests in queue |
| display\_course\_allocations() | O(n) | Displays all course allocations |

Note: list.pop(0) for dequeue is O(n) in Python, but we only do the entire queue once.

Module 3: Fee Tracking (Binary Search Tree)

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| Add payment record() | O(log n) average, O(n) worst | BST insertion with tree balancing |
| Search payment record() | O(log n) average, O(n) worst | BST search |
| Update payment record() | O(log n) average, O(n) worst | Search + update |
| Generate fee clearance report() | O(n) | Inorder traversal visits all nodes |

Balancing Concern: No balancing, worst-case degrades to O(n) for unbalanced trees.

Module 4: Library System (Hash Map)

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| add\_book() | O(1) average | Hash-based insertion |
| borrow\_book() | O(1) average | Hash-based access + list append |
| return\_book() | O(1) average | Hash-based access + list removal |
| check\_availability() | O(1) average | Direct hash lookup |

Module 5: Performance Analytics (Max Heap)

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| add\_performance\_record() | O(log n) | Heap insertion with percolate up |
| display\_top\_performer() | log n) | Extract max k times |
| view\_all\_rankings() | O(n log n) | Heap sort equivalent |

**2. Space Complexity Analysis**

Overall System: O(n + m + p + q + r)

Where:

* n = number of students
* m = number of course registration requests
* p = number of payment records
* q = number of books
* r = number of performance records

Individual Modules:

|  |  |  |
| --- | --- | --- |
| Module | Space complexity | Components |
| Student Registry | O(n) | Hash table of n student records |
| Course Scheduling | O(m) for queue + O(c) for allocations | Queue + course-student mappings |
| Fee Tracking | O(p) | BST with p nodes, each containing payment details |
| Library System | O(q) | Hash table of q book records with borrower lists |
| Performance Analytics | O(r) | Heap array + hash map for detailed score |

Memory Breakdown per Data Structure:

- Hash Table Entry: ~100-200 bytes (keys, values, overhead)

- BST Node: ~40-80 bytes (pointers + data)

- Heap Element: ~20-40 bytes (score + reference)

- Queue Element: ~20-30 bytes (tuple data)

**3. Trade-offs and Design Alternatives**

Module 1: Student Registry - Hash Table vs. Alternatives

Chosen: Hash Table

- Advantage: O(1) average lookup, perfect for frequent searches

- Disadvantage: No ordering inherent, memory overhead of hashing

Alternative: Balanced BST (AVL/Red-Black)

- Pros: Ordered traversal, better worst-case performance

- Cons: O(log n) lookup vs O(1), more complex implementation

- Decision: Chose hash table because of high lookup frequency of students and rare ordering.

Module 2: Scheduling Courses - Queue vs. Alternatives

Chosen: Simple Queue (List)

- Pros: Simple to implement, natural FIFO

- Cons: O(n) dequeue with pop(0)

Alternative: collections.deque

- Pros: O(1) enqueue and dequeue

- Cons: Slightly more complex API

- Decision: Chose list due to simplicity since queue size is manageable

Module 3: Tracking Fees - BST vs. Alternatives

Chosen: Binary Search Tree

- Pros: Natural report ordering, range queries efficient

- Cons: Worst-case O(n) if no balancing

Alternative: Sorting + Hash Table

- Pros: O(1) lookup, simpler to implement

- Cons: O(n log n) to build sorted reports

- Decision: Went with BST because fee reports require sorted output by student ID

Module 4: Library System - Hash Table vs. Alternatives

Chosen: Hash Table

- Pros: Perfect for ISBN-based lookup, O(1) operations

- Cons: No implicit ordering of books

Alternative: Balanced BST

- Pros: Ordered list of books

- Cons: O(log n) lookups as opposed to O(1)

- Decision: Hash table is perfect for ISBN-based access patterns

Module 5: Performance Analytics - Max Heap vs. Alternatives

Chosen: Max Heap

- Pros: O(1) access to top performer, ranking efficient

- Cons: No efficient random access

Alternative: Sorted Array

- Pros: O(1) random access, easy implementation

- Cons: O(n log n) insertion compared to O(log n) for heap

- Decision: Max heap is perfect for top-performer queries that are frequent

**4. Optimality Justification**

High-Frequency Operations Optimized:

- Student lookups (hash table): O(1) vs O(log n)

- Library book searches: O(1) vs O(log n)

- Top performer access: O(1) vs O(n)

Memory vs. Speed Trade-offs:

- Used hash tables where speed crucial (registries, library)

- Used BST where sorted data needed (fee reports)

- Used heap where ranking important (performance)

Real-World School Usage Patterns:

- Frequent: Student searches, book checkouts

- Moderate: Fee updates, course registrations

- Infrequent: Detailed reports, rankings

Scalability Considerations:

- Hash tables scale to thousands of students easily

- BST performs well for medium dataset sizes

- Heap performant for typical class sizes (20-200 students)

**5. Potential Improvements**

- BST Balancing: Employ AVL or Red-Black tree for O(log n) guaranteed operations

- Queue Optimization: Use collections.deque for O(1) dequeueing

- Caching: Cache frequently accessed reports

- Persistence: Use database backend for large installations

**Conclusion**

The data structures chosen provide the optimal trade-off for a school management system:

- Fast access for common operations (O(1) where possible)

- Reasonable memory usage (O(n) across modules)

- Trade-offs appropriately among the different performance characteristics

- Scalable design that can accommodate typical school sizes successfully