ASSIGNMENT 2

DSE-01-8283/2023

DATA STRUCTURES AND ALGORITHMS

DISCUSSION QUESTIONS

1.How did the use of arrays as a data structure impact ABC Tech's performance before optimization?

Due to unsorted data their search operations required linear search with O(n) time complexity which caused searches to slow down when the dataset expanded.

The process of record retrieval became slow because the system lacked an effective indexing technique which forced users to scan through the entire array.

The fixed array size results in space consumption problems because arrays either use extra resources unnecessarily or force users to adjust their sizes frequently thus affecting system performance.

Task execution slowed down when lists and queues received suboptimal application since managers performed scheduling duties one by one resulting in performance decrements.

2.Why was binary search chosen over sequential search for the new system?

Time Complexity Improvement:

Sequential Search function has an O(n) time complexity because it examines every element one by one until it identifies the target value.

Turnaround time decreases with log n exponent for this search method because a single step divides the dataset into two sections.

Scalability:

Linear search stops being practical for increasing datasets because it requires an increasing number of element-to-element comparisons.

Large datasets can be managed efficiently by the logarithmic efficiency of binary search.

Faster Lookups:

The practice of binary search enables lightning-fast searches which brings better performance for complete systems.

Optimization with Sorted Data:

The predictive search times afforded by binary search become possible because it needs sorted arrays either directly or through binary search trees.

Trade-off:

For binary search to operate the fundamental requirement is sorted data but the data needs an initial sorting operation (O(n log n)). Secular search becomes more efficient than sequential search after data has been properly ordered.

3.Explain how priority queues improved task scheduling efficiency.

. Faster Task Selection

Before Optimization (Regular Queue / List)

Tasks followed a FIFO structure so that newer urgent matters needed to wait until all prior tasks finished being processed.

A standard linear scan operation (O(n)) slowed down the process of finding the highest-priority task.

After Optimization (Using a Priority Queue / Heap)

A min-heap or max-heap served as a priority queue system which delivered the most critical task within O(log n) complexity.

The system ceased its full list scanning procedure for locating the top urgent task.

2. Reduced Processing Delays

The priority queue system enabled critical system updates together with urgent user requests to get processed before tasks with lower priority.

The system reduced its inactive periods and reacted more rapidly for urgent operations.

3. Efficient Task Insertion and Removal

Casting new tasks into the priority queue required O(log n) time thus being shorter than manual reorganization every time.

The queue system provided O(log n) performance for both processing queues to completion and removing tasks.

4.Compare the effectiveness of merge sort versus quick sort in ABC Tech's data organization.

Comparison of Merge Sort vs. Quick Sort in ABC Tech's Data Organization

The data organization at ABC Tech received support from Merge Sort along with Quick Sort. Sorting algorithms exhibit distinct features that become important based on the type of data to be sorted.

1. Time Complexity

Algorithm Best Case Average Case Worst Case

The performance of Merge Sort remains O(n log n) in all possible scenarios of usage.

The running time for Quick Sort becomes $O(n^2)$ as a result of poor partition decisions.

Within all execution scenarios Merge Sort demonstrates a consistent performance at O(n log n) complexity level.

The choice of poor pivot elements during Quick Sort execution will lead to $O(n^2)$ complexity behavior in arrays that are almost sorted when the smallest or largest elements become pivot points.

2. Space Complexity

Algorithm Space Complexity

Merge Sort O(n) (requires extra memory)

Quick Sort O(log n) (in-place sorting)

The need for extra memory storage in Merge Sort operations diminishes its efficiency when using available system resources.

As an in-place sorting method Quick Sort operates directly from memory which proves advantageous when you must minimize the use of additional storage space.

3. Performance on Different Data Types

Scenario Better Algorithm Why?

Small datasets Quick Sort Faster due to low overhead

Large datasets Merge Sort More consistent performance

The speed of Quick Sort algorithm decreases to $O(n^2)$ when sorting nearly sorted data.

Random data requires the Quick Sort algorithm to function efficiently due to its effective pivot selection mechanism.

The merge sorting algorithm works efficiently with linked lists because they offer seamless merging capabilities.

4. ABC Tech's Use Case

The data conditions at ABC Tech determine the selection between Merge Sort and Quick Sort because the company handles big datasets together with complex query structures.

The application of Quick Sort focused on general sorting needs because the system required memory optimization.

Large datasets of linked lists benefited from Merge Sort because it provided stable performance at O(n log n).

5.How did implementing hash tables enhance data retrieval speed?

1. Fast Lookup Time (O(1) Average Case)

Before optimization ABC Tech worked with unsorted arrays that needed linear search searching with a running time of O(n).

The hash table allows data storage through key-value pairs which enables O(1) average time lookup through direct index-based retrieval.

The search time dramatically shortened down to almost immediate response.

Example:

The process of linear search in 1 million records to locate a user ID would require up to 1,000,000 search operations.

Tests have revealed that hash tables require only one operation to locate the same record when the conditions are optimal.

2. Efficient Data Organization and Storage

Data mappings through hashing allow instant access to stored information at specific indexes.

Hash tables implement chaining or open addressing methods to guarantee high efficiency when multiple records result in the same index.

3. Improved Performance in Database Queries

Hash tables improved database indexing capabilities thus enabling faster executions of database

queries.

A customer’s order history search performance improved dramatically for improved efficiency.

4. Optimized Caching and Session Management

The storage of standard data that users accessed often used hash tables including user sessions and system configurations.

The system response improved because repeated database calls became minimized.

6.What role did graph algorithms play in optimizing network routing?

1. The mathematical abstraction of the network takes the form of a visual representation using graph principles.

The infrastructure of servers routers and connections at ABC Tech became a graphical representation which contained the following elements:

The network devices alongside servers make up the set of graph vertices.

Network nodes function as vertices and edges demonstrate the device connections where latency measurements and bandwidth expense values serve as edge weights.

Load Balancing with Graph Algorithms

The network connections reached their minimum total cost through Minimum Spanning Tree (MST) using Prim’s or Kruskal’s Algorithm.

Even distribution of traffic flow helped prevent congestion from forming at vital network nodes.

4. Detecting and Preventing Network Failures

Bellman-Ford Algorithm acted as a mechanism to detect negative weight cycles which then prevented the formation of routing loops that could lead to network failure.

Through Floyd-Warshall Algorithm companies obtained the shortest paths between any pair of servers which provided backup network routes when servers failed.

7.Discuss the time complexity differences between linear search and binary search.

1. Linear Search

Definition:

This method looks for elements by making one sequential inspection across the entire list.

It performs the same algorithm for organized and disorganized list sets.

Time Complexity:

Case Time Complexity

The Element searching process takes O(1) when it finds the target element at the very beginning.

Average Case (Element in the middle) O(n)

The linear search operation takes O(n) time to finish when the element exists after the first item or not present at all.

Example:

Finding 50 in [10, 20, 30, 40, 50]:

The search stops at the value 50 that sits in position 4 (fifth position) of the list.

This method needs n comparisons during its most unfavorable scenario.

☑ When to Use:

Small datasets.

Unsorted or unordered data.

The algorithm is appropriate for data that experiences many insertions or deletions operations because sort operations are not necessary.

2. Binary Search

Definition:

The technique divides a sorted array into two equal parts before conducting the search operation within the appropriate section.

Sorting of the data must occur before conducting the search.

Time Complexity:

Case

Time Complexity

The time consumption for this search algorithm amounts to O(1) when finding the element in the middle portion of the data range.

Average Case O(log n)

Example:

Finding 50 in [10, 20, 30, 40, 50]:

The search continues within the right section because 30 is the pivot point at index 30 and 50 exceeds it.

The first middle point at 40 indicates a right-side search direction since 50 is greater than it.

The algorithm found 50 during its third comparison (against the binary search requirement of five comparisons in linear search).

When to Use:

O(log n) outperforms O(n) in processing large datasets because it operates at speed levels that are faster than n.

When data is already sorted.

For fast lookups in databases and search engines.

3. Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Key Difference:

Linear Search requires O(n) time to perform its searches therefore the search duration increases linearly with the input size.

The search time of Binary Search grows extremely slow as the input size escalates since it operates at an O(log n) complexity.

4. Real-World Impact

Linear Search: Up to 1,000,000 comparisons

Binary Search: Only ~20 comparisons ($\log_2(1{,}000{,}000) \approx 20$)

8.Why is it important to understand Big-O notation when analyzing algorithm efficiency?

Learning about Big-O Notation proves vital during efficiency analysis of algorithms.

1. Measures Algorithm Performance

The analysis of algorithm runtime increases and space requirements through Big-O notation gets possible as input sizes become larger.

The notation enables a standard method for comparing multiple algorithms against one another.

For large data sets the use of linear search (O(n)) turns out slower than binary search (O(log n)) because the binary search algorithm works at an exponential speed.

2. Predicts Scalability

The ability to forecast algorithm performance growth with big data datasets becomes possible by understanding big-O principles.

O(log n) or O(n) → Good for large inputs.

The algorithms performance slows to an unacceptable point when data quantities grow because of $O(n^2)$ and $O(2^n)$ time complexity.

Example:

Sorting 10 million records:

Quick Sort achieves faster execution compared to Bubble Sort since it operates at O(n log n) versus its $O(n^2)$ speed.

3. Operating systems can thrive better by minimizing their Time and Space requirements through Big-O optimization.

Big-O serves developers to identify the most efficient algorithm which leads to reduced:

Execution time (speed).

Memory consumption (space complexity).

Computational cost (server resources).

Example:

Searching in a database:

Hash Table (O(1)) → Instant lookup.

The search method Binary Search operates at O(log n) but maintains its speed only after database sorting occurs.

Linear Search (O(n)) → Slow for large databases.

4. Helps in Choosing the Right Algorithm

Knowing Big-O classification enables developers to select the optimal algorithm for particular problems they need to solve.

Graph problems → Use Dijkstra's Algorithm (O((V + E) log V)) instead of Brute Force ($O(V^2)$).

Large dataset sorting requires merge sort algorithm (O(n log n)) instead of bubble sort ($O(n^2)$).

5. Avoids Performance Bottlenecks

Failure becomes likely when analyzing programs without Big-O methods because the program works on smaller datasets but not larger ones.

Example:

A social media application requires a username to run a search operation.

A search utilizing Linear Search (O(n)) functions on a slow pace when millions of users need to be searched.

Large-scale applications benefit best from Hash Tables since they deliver O(1) performance for instant searches.

9.How can ABC Tech further enhance their system using AVL trees or B-trees?

1. AVL Trees help speed up searching operations.

The self-balancing binary search tree AVL Tree maintains subtrees with balance factor differences that never exceed one unit (1).

How It Helps ABC Tech:

The balance factor in the AVL tree guarantees that search and insert and delete operations take O(log n) time.

An AVL Tree maintains O(log n) time performance while a regular BST might decline to O(n) like a linked list.

The tree structure maintains efficiency during both searches and insertions because it stays balanced.

Example Use Case:

User passwords stored with AVL trees allow immediate and efficient credential retrieval to speed up authentication.

The database uses this technique to add indexes which makes search queries execute rapidly.

2. Database and file system indexing benefits from the implementation of B-Trees.

Large-scale databases together with disk-based storage systems benefit from the self-balancing B-Tree structure because it functions as a multi-way search tree.

How It Helps ABC Tech:

B-Trees optimize storage performance because they minimize the number of disk access requirements in large database implementations.

B-Trees excel with expansive datasets since their depth-minimizing structure makes both search operations and insertion and deletion processes highly efficient.

The data structure preserves logarithmic height through its balanced arrangement which brings efficient read/write operation speeds.

B-Trees serve as indexing structures in both database systems and file systems management since MySQL along with MongoDB databases among others implement them for their query optimization.

Example Use Case:

B-Trees allow organizations to store millions of product records for fast lookup operations during e-commerce order processing.

Cloud storage systems deploy effective file retrieval methods which operate in large-scale storage environments.

3. Comparing AVL Trees vs. B-Trees

Feature AVL Tree B-Tree

B-Trees suit both in-memory search operations and storage needs for extensive databases and disk storage applications.

Search Time O(log n) O(log n)

The insertion or deletion of elements in AVL trees requires O(log n) time along with rebalancing operations which B-trees need less rebalancing operations to achieve an O(log n) time.

Tree Structure Strictly balanced (binary) Multi-way balanced

The storage of balance factors consumes more memory while disk storage utilizes less memory.

QUESTION B

```
#include <stdio.h>
#include <stdlib.h>

// Structure to represent an order
typedef struct Order {
    int orderID;
    struct Order* next;
} Order;

// Structure to represent the order queue
typedef struct OrderQueue {
    Order* front;
    Order* rear;
} OrderQueue;

// Function to create an empty queue
OrderQueue* createQueue() {
    OrderQueue* q = (OrderQueue*)malloc(sizeof(OrderQueue));
    q->front = q->rear = NULL;
    return q;
}
```

```
// Function to add a new order to the queue
void enqueue(OrderQueue* q, int orderID) {
    Order* newOrder = (Order*)malloc(sizeof(Order));
    newOrder->orderID = orderID;
    newOrder->next = NULL;

    if (q->rear == NULL) {  // If queue is empty
        q->front = q->rear = newOrder;
        return;
    }

    q->rear->next = newOrder;
    q->rear = newOrder;
    printf("Order %d added to the queue.\n", orderID);
}

// Function to cancel an order from the queue (by order ID)
void cancelOrder(OrderQueue* q, int orderID) {
    if (q->front == NULL) {
        printf("No orders in the queue to cancel.\n");
        return;
    }

    Order* temp = q->front, *prev = NULL;

    // If the order to cancel is at the front
    if (temp != NULL && temp->orderID == orderID) {
        q->front = temp->next;
        if (q->front == NULL)  // If queue becomes empty
            q->rear = NULL;
```

```
        free(temp);
        printf("Order %d has been canceled.\n", orderID);
        return;
    }

    // Search for the order in the queue
    while (temp != NULL && temp->orderID != orderID) {
        prev = temp;
        temp = temp->next;
    }

    // If order not found
    if (temp == NULL) {
        printf("Order %d not found.\n", orderID);
        return;
    }

    // Remove order from the queue
    prev->next = temp->next;
    if (temp == q->rear)  // If last order is removed
        q->rear = prev;
    free(temp);
    printf("Order %d has been canceled.\n", orderID);
}

// Function to display all pending orders
void displayOrders(OrderQueue* q) {
    if (q->front == NULL) {
        printf("No pending orders.\n");
        return;
```

```
    }

    Order* temp = q->front;
    printf("Pending Orders: ");
    while (temp != NULL) {
        printf("%d -> ", temp->orderID);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to process orders (FIFO)
void processOrder(OrderQueue* q) {
    if (q->front == NULL) {
        printf("No orders to process.\n");
        return;
    }

    Order* temp = q->front;
    printf("Processing Order %d...\n", temp->orderID);
    q->front = q->front->next;
    if (q->front == NULL)  // If queue becomes empty
        q->rear = NULL;
    free(temp);
}

// Main function to test the order system
int main() {
    OrderQueue* q = createQueue();
```

```
    enqueue(q, 101);
    enqueue(q, 102);
    enqueue(q, 103);
    displayOrders(q);

    cancelOrder(q, 102);
    displayOrders(q);

    processOrder(q);
    displayOrders(q);

    processOrder(q);
    displayOrders(q);

    return 0;
}
```

OUTPUT

```
Order 101 added to the queue.
Order 102 added to the queue.
Order 103 added to the queue.
Pending Orders: 101 -> 102 -> 103 -> NULL
Order 102 has been canceled.
Pending Orders: 101 -> 103 -> NULL
Processing Order 101...
Pending Orders: 103 -> NULL
Processing Order 103...
No pending orders.
```