DATA STRUCTURES AND ALGORITHMS

Nguyen Quang Linh
BH01804

INDEX

1. INTRODUCTION

Overview of Data Structures and their Importance

- Data structures are the foundation of data organization and management, allowing for more efficient operations in programs and systems.
- They define how data is stored, processed, and retrieved, and play an important role in almost every software application.

Importance:

- Optimizing performance: Choosing the right data structure helps optimize time and space, making the program run faster and consume less resources.
- Solving complex problems: Problems ranging from sorting, searching to calculating the shortest path in a network rely on efficient data structures and algorithms.
- Wide applications: From artificial intelligence, data science, embedded systems to database management, data structures appear in many different fields.

2. DATA STRUCTURE SPECIFICATION DESIGN

IDENTIFYING DATA STRUCTURES

1. Objective: Classify and explain basic data structures.

2. Expanded content:

- Array: Static arrays, dynamic arrays and index types.
- Linked List: Single, double, circular linked lists. Advantages over arrays (flexible insertion and deletion).
- Stack: Stacks are used in tree traversal, recursive processing.
- Queue: Regular queue and priority queue.
- Tree: Binary tree, binary search tree (BST), AVL tree.
- Hash Table: Conflict handling techniques, such as linear probing, double hashing.

DEFINITION OF OPERATIONS

- **Insert: How to insert in dynamic arrays and linked lists.**
- **Delete: Difference between deleting at the beginning/end and deleting at any position in linked lists and arrays.**
- **Search: Linear Search (O(n)) and Binary Search (O(log n)).**
- **Sort: Popular sorting algorithms, from simple (Bubble Sort) to complex (Merge Sort, Quick Sort).**
- **Traverse: Traverse arrays and trees in pre-order, in-order, and post-order.**

INPUT PARAMETERS

For each operation on a data structure, it is necessary to specify the input parameters to ensure that the operations are performed correctly. The parameters include the format, data type, and specific location for the operations.

1. INSERT INTO ARRAY:
REQUIRED PARAMETERS:
- **VALUE:** THE VALUE TO BE INSERTED INTO THE ARRAY. THIS CAN BE AN INTEGER, A STRING, OR ANY OTHER DATA TYPE DEPENDING ON THE ARRAY.
- **INDEX:** THE INDEX IN THE ARRAY WHERE THE NEW VALUE WILL BE INSERTED.

2. DELETE FROM LINKED LIST:
REQUIRED PARAMETERS:
- **INDEX:** INDEX OF THE ELEMENT TO BE DELETED (USED FOR SINGLY OR DOUBLY LINKED LISTS).
- **REFERENCE:** DIRECT REFERENCE TO THE ELEMENT TO BE DELETED (CAN BE A POINTER OR A SPECIFIC OBJECT).

3. PUSH INTO STACK:
REQUIRED PARAMETERS:
- **VALUE:** THE VALUE TO BE PUSHED INTO THE STACK. THE DATA TYPE CAN BE ANY (INTEGER, STRING, OBJECT).

4. ENQUEUE INTO QUEUE:
REQUIRED PARAMETERS:
- **VALUE:** THE VALUE TO BE ENQUEUED. THIS CAN BE A NUMERIC, STRING, OR OBJECT DATA TYPE DEPENDING ON THE SPECIFIC APPLICATION.

CONDITIONS BEFORE AND AFTER OPERATION

EXPLANATION OF PRE-CONDITIONS AND POST-CONDITIONS:

- Pre-condition: Are conditions that must be satisfied before performing an operation.
- Post-condition: Are conditions that must be satisfied after performing an operation.

Example

OPERATION: INSERT ELEMENT INTO ARRAY

Pre-condition:

- The array must have enough capacity (for static arrays) before adding a new element.
- For dynamic arrays, if the current size is equal to the maximum capacity, the array needs to be allocated more memory to accommodate the new element.

Post-condition:

- The new element must be added at the correct specified position in the array.
- If it is a sorted array, after insertion, all elements must be rearranged in the correct order.

TIME AND MEMORY COMPLEXITY

Big O Notation Explained

Big O notation helps evaluate the performance of algorithms, showing how they perform as the input size increases. It describes how the time or memory required increases as the input data increases, which helps us choose the right data structure and algorithm.

- O(1): Immediate operations such as accessing elements in an array.
- O(n): Traversing a linked list or array.
- O(log n): Traversing a binary search tree.
- O(n log n): Merge Sort, Quick Sort.
- O(n^2): Simple sorting algorithms such as Bubble Sort.

3. MEMORY STACK IN FUNCTION CALL

MEMORY STACK DEFINITION

1. Concept:

- Memory Stack is an important data structure in programming, especially in systems programming. Stacks are used to manage memory for local variables, store program state, and make function calls during execution.

2. Functions and Roles of Memory Stacks

- Memory Management
- Function Calling
- Recursion Management

3. Benefits of Stack Memory

- Fast Access Speed: Accessing and managing memory in the stack is faster than in the heap (dynamic memory) due to the simple management mechanism.
- Automatic Release: When the function completes, the corresponding stack frame is automatically released, reducing the risk of memory leaks.

OPERATIONS ON MEMORY STACK

- The memory stack supports several basic operations that help manage stack frames during program execution. Here are the three main operations: Push, Pop, and Peek.

PUSH: ADD NEW STACK FRAME

- When a function is called, a new stack frame is created and added to the top of the stack.
- This frame contains the function's parameters and return address, helping the program know where to continue when the function ends.

POP: REMOVING A STACK FRAME

- When a function completes, the corresponding stack frame is removed (pop) from the top of the stack.
- This frees up memory and returns control to the calling function.

PEEK: VIEW INFORMATION WITHOUT PERFORMING AN OPERATION

- The Peek operation allows you to view the return address or parameters of a function without removing the stack frame.
- This is useful when you need to examine information without changing the state of the stack.

HOW TO MAKE FUNCTION CALLS USING STACKS

1. When Calling a Function: Push Operation

a. Storing the Return Address:

- As soon as a function is called, the address that the program will return to after the function completes is pushed onto the stack. This ensures that the program knows where to continue from.

b. Storing Function Parameters:

- All parameters passed to the function will also be stored in a new stack frame. Each parameter is pushed onto the stack in order from left to right.

c. Storing Local Variables:

- All local variables of the function will be initialized and stored onto the stack. This allows the function to access and modify these variables during execution.

2. Function Execution

- When the function is called, the program begins executing the statements inside the function. At this point, the program can access the parameters and local variables stored on the stack.

3. When the Function Finishes: Pop Operation

a. Release the Stack:

- When the function completes, the corresponding stack frame is removed (pop) from the stack. This frees up the memory allocated for the parameters and local variables.

b. Return Address Access:

- The return address from the stack frame is retrieved, and the program continues from that address, back to the point where the function was called.

STACK FRAME AND ITS ROLE

1. Stack Frame Concept

A stack frame is an important structure in the memory stack that stores all the information needed to execute a function call. Each time a function is called, a new stack frame is created to hold the function's local variables, parameters, and return address.

2. Structure of a Stack Frame

a. Base Pointer (BP):

- BP is used to store the address of the bottom of the current stack frame. It holds information about the location of parameters and local variables in the stack frame.

b. Stack Pointer (SP):

- SP points to the top of the stack and moves up and down as stack frames are created or released. When a function is called, SP is incremented to store the new stack frame. When the function ends, SP is decremented as the stack frame is removed.

3. Role of Stack Frame

- Local Variable and Parameter Management
- Keeping Call State
- Supporting Recursion
- Structure Maintenance

THE IMPORTANCE OF MEMORY STACKS

The memory stack plays a crucial role in handling recursive functions. When a function calls itself, the memory stack helps keep track of function calls and maintains the state of each call, thus allowing for correct and efficient execution.

4. FIFO QUEUE

INTRODUCING FIFO QUEUE

1. Definition of FIFO Queue

A FIFO (First In, First Out) queue is a data structure that allows elements to be added and removed on a “first in, first out” basis. This means that the first element added to the queue will be the first element removed. A FIFO queue works similar to a queue, where the person at the head of the queue will be served first.

2. FIFO Queue Structure:

- Enqueue: New element is added to the tail of the queue.
- Dequeue: Element is removed from the head of the queue

3. Applications of FIFO Queues

- Printing Systems
- CPU Scheduling
- Data Transmission Networks
- Resource Management

QUEUE STRUCTURE

1. Structure of Queue

A queue is a linear data structure that allows the management of elements according to the FIFO (First In, First Out) principle.

2. Queues have two main operations:

- Enqueue
- Dequeue

3. Queues can be implemented in many different ways, including:

- Array-Based Queues
- Linked List-Based Queues

EXAMPLE: FIFO QUEUE

1. Operating Principle:

- When a user sends a print command, it will be added to the print queue (Enqueue).
- The printing system will take the first command in the queue to execute (Dequeue).

2. Process:

- Step 1: User A sends a command to print document A.
- Step 2: User B sends a command to print document B.
- Step 3: User C sends a command to print document C.
- The print queue will now have the order: [A, B, C].

3. Processing Print Commands:

- Step 4: The system takes the first print command (document A) out of the queue to print.
- Step 5: When document A finishes printing, it will be removed from the queue.
- Step 6: Next, the print command for document B will be processed.
- The print queue will now have the following order: [B, C].

4. Result:

- Printed document A will be completed first, then document B and finally document C, keeping the FIFO principle.

```python
class PrintQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, document):
        self.queue.append(document)
        print(f"Document {document} added to the print queue.")

    def dequeue(self):
        if self.queue:
            return self.queue.pop(0)
        else:
            return None

    def print_documents(self):
        while self.queue:
            document = self.dequeue()
            print(f"Printing document: {document}")
        print("All documents have been printed.")

# Khởi tạo hàng đợi in
print_queue = PrintQueue()

# Thêm lệnh in
print_queue.enqueue("Document A")
print_queue.enqueue("Document B")
print_queue.enqueue("Document C")

# Bắt đầu in tài liệu
print_queue.print_documents()
```

PERFORMANCE COMPARISON: ARRAYS VS. LINKED LISTS

Characteristics	Array	Links List
Access	Fast (O(1))	Slower (O(n))
Size	Fixed, can be wasteful	Flexible, unlimited
Resources	Takes up less memory if fixed	Takes up more memory
Add/Delete Operations	Difficult (O(n) if moving is required)	Easy (O(1))

5. COMPARISON OF SORTING ALGORITHMS

INTRODUCTION TO SORTING ALGORITHMS

1. Merge Sort

a. Definition:

- Merge Sort is a divide and conquer sorting algorithm. It divides the list to be sorted into two halves, sorts each halves, and then merges them into a sorted list.

b. Process:

- Divide: The list is divided into two halves until each half has only one element.
- Sort and Merge: The halves are merged during the sorting process.

c. Application:

- Suitable for sorting large data, such as sorting files or databases.
- Used in applications that require stability, where the order between equal elements needs to be maintained.

2. Quick Sort

a. Definition:

- Quick Sort is also a divide and conquer sorting algorithm, but it works by selecting an element as a "pivot" and dividing the list into elements smaller and larger than the pivot. The algorithm is then applied recursively to the elements to the left and right of the pivot.

b. Process:

- Pivot Selection: Select an element as a pivot (usually the first, middle, or last element).
- Divide: Reorder the list so that all elements smaller than the pivot are on the left, and all elements larger than the pivot are on the right.
- Recursive: Apply Quick Sort to the left and right elements.

c. Application:

- Suitable for in-memory sorting.
- Widely used in applications that require high performance and low memory.

TIME COMPLEXITY COMPARISON

Algorithm	Best Case	Average Case	Worst Case
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

- **Merge Sort:** Time complexity is stable at O(n log n) in all cases.
- **Quick Sort:** Fast at O(n log n) in average and best case, but can be slow at $O(n^2)$ in worst case.

MEMORY COMPLEXITY COMPARISON

Algorithm	Memory Complexity	Detail
Merge Sort	O(n)	An extra array is required to store the elements during the merge, which consumes more memory.
Quick Sort	O(log n) (average), O(n) (worst)	Only stack memory is needed for recursive calls, which is less memory intensive under ideal conditions.

- **Merge Sort:** Requires extra memory space equal to the size of the original array (O(n)) to store the elements when merging. This can be memory intensive if the data size is large.
- **Quick Sort:** Consumes less memory, as it only needs to store the recursive states in the stack (O(log n) in the average case). However, in the worst case, the stack memory can increase to O(n) when choosing an unreasonable pivot.

STABILITY

Algorithm	Stability	Detail
Merge Sort	Stable	Elements with equal values retain their original order after sorting.
Quick Sort	Unstable	The order of elements with equal values may change after sorting, depending on how the pivot is chosen.

- **Merge Sort:** Stable because it always maintains the original order of equal elements during the merge. This is useful when it is important to maintain the original order in data.
- **Quick Sort:** Unstable because the process of selecting pivots and swapping elements can change the order of equal elements. This may not be important in some applications, but is a factor to consider in special situations.

COMPARISON TABLE

Criteria	Merge Sort	Quick Sort
Time Complexity	O(n log n) (best, average, worst)	O(n log n) (average), $O(n^2)$ (worst)
Memory Complexity	O(n) (due to need for sub-array)	O(log n) (average), O(n) (worst)
Stability	Stable	Unstable
Practical Applications	Arrange large files, data needs to be stable	In-memory sorting, high performance

- **Merge Sort:** Preferred for applications that require stability and large memory space. Often used when processing peripheral data such as large files or databases.
- **Quick Sort:** Often used for in-memory data (in-memory sorting) with the advantage of speed and memory saving. However, be careful with the worst case that can cause poor performance.

6. SHORTEST PATH ALGORITHM

INTRODUCTION TO SHORTEST PATH ALGORITHM

Shortest Path: The problem of finding the shortest path between two vertices in a weighted graph. Shortest path algorithms help optimize the search for optimal paths through complex networks such as:

- **Transportation**: Finding the fastest path between two points in a transportation system.
- **Computer networks:** Finding the most efficient route to transmit data through computer networks.

Applications:

- **Route planning:** Used in mapping applications, GPS to find the fastest route.
- **Network optimization:** Used to route data through computer networks to reduce latency and maximize bandwidth.
- **Graph search:** Analyzing data, such as social networks or connecting cities.

Popular Algorithms:

- Dijkstra's Algorithm.
- Prim-Jarnik Algorithm.

ALGORITHM 1: DIJKSTRA'S ALGORITHM

a. Goal: Find the shortest path from a source vertex to all other vertices in a non-negative weighted graph.

b. How it Works:

- The algorithm starts at the source vertex, marking its distance as 0.
- Continues to traverse the weighted adjacent vertices, updating the shortest distance to each unvisited vertex.
- Choose the unvisited vertex with the smallest distance, repeating the process until all vertices are processed.

Complexity:

- **Time:** $O(V^2)$ for basic implementation, or $O(E + V \log V)$ for priority queue implementation.
- **Memory:** $O(V)$ for storing information about vertices and shortest distances.

Applications:

- **GPS navigation systems:** Find the fastest route between locations.
- **Computer networks:** Find the optimal route for data transmission.
- **Path planning:** Applications in games, robot navigation.

ALGORITHM 2: PRIM-JARNIK

Objective: Prim-Jarnik Algorithm (also known as Prim's Algorithm) finds the Minimum Spanning Tree (MST) in a weighted graph. The minimum spanning tree is a set of edges in the graph that connect all vertices with the smallest total weight without forming a cycle.

How it Works:

- Start at any vertex (usually the source vertex).
- Select the edge with the smallest weight connecting the current vertex to an unvisited vertex.
- Add this edge to the minimum spanning tree and continue to traverse the adjacent edges from the visited vertex set.
- Repeat the process until all vertices in the graph are added to the spanning tree.

Complexity:

- **Time:** $O(V^2)$ with basic implementation, or $O(E \log V)$ if using priority queue.
- **Memory:** $O(V)$ to store vertex information and minimum spanning tree.

Applications:

- **Communication networks:** Optimize the connection of communication networks at the lowest cost.
- **Circuit design:** Create connections between nodes in the circuit with the shortest total cost.
- **Transportation:** Build optimal road and railway systems, minimizing construction costs.

PERFORMANCE ANALYSIS AND COMPARISON

Criteria	Dijkstra's algorithm	Prim-Jarnik algorithm
Target	Find the shortest path from a source vertex to all other vertices in the graph.	Find the minimum spanning tree that connects all vertices in the graph.
Graph Type	The graph has non-negative weights.	The graph has arbitrary weights (non-negative or negative).
Time Complexity	$O(V^2)$ or $O(E + V \log V)$ with priority queue.	$O(V^2)$ or $O(E \log V)$ with priority queue.
Application	GPS systems, computer networks, route planning.	Communication network, circuit design, connection optimization.
Features	Find the shortest path from one vertex to multiple vertices.	Build the smallest tree span at the lowest total cost.

PERFORMANCE ANALYSIS AND COMPARISON

Dijkstra:

Advantages: Finds the shortest path from a vertex to all other vertices in a graph, suitable for routing problems and navigation systems.

Disadvantages: Only works well for graphs with non-negative weights. Not optimal for finding connections to the entire graph.

Prim-Jarnik:

Advantages: Finds the smallest spanning tree, optimal when connecting all vertices in a graph with the smallest cost. Often used in network design, where the minimum cost of connection is required.

Disadvantages: Does not solve the shortest path problem for a specific vertex.

THANK YOU

Nguyen Quang Linh
BH01804