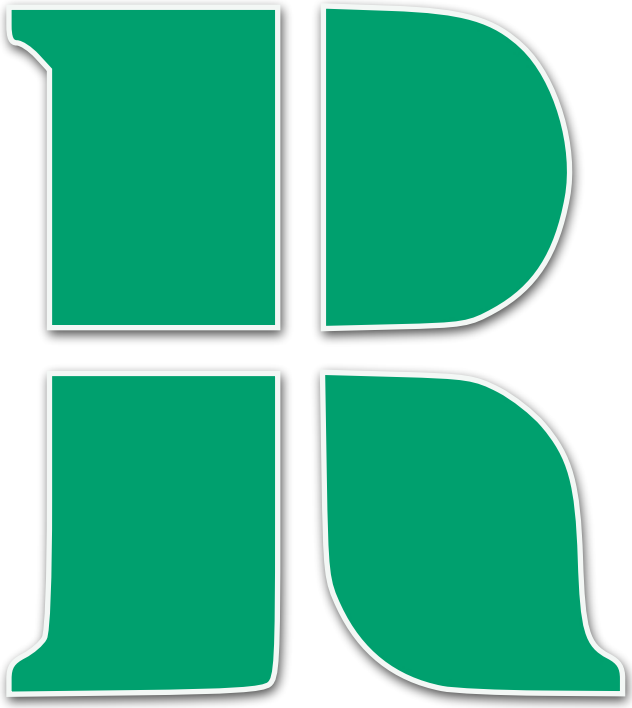


COURSE_WORK
On
ALGORITHM

SUBMITTED BY:

NAME	STUDENT ID
BHOWAN KHAWAS	A00028871

FULL REPORT CONTENT (Tasks 1, 2, & 3)

TASK 1: Sorting Algorithms

(a) Merge sort

Input Data: ['b', 'h', 'o', 'w', 'a', 'n', 'k', 'h', 'a', 'w', 'a', 's']

Algorithm Breakdown:

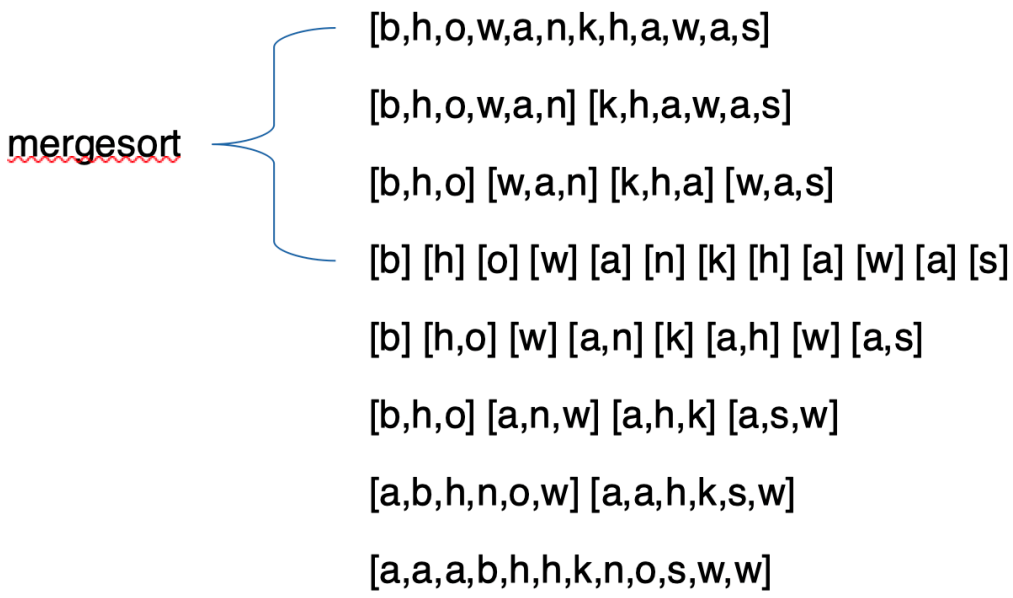
Merge sort is an efficient sort algorithm that employs the “Divide and Conquer” technique. Its algorithm is recursive and proceeds as below:

- Divide: The input array is divided into two halves until both sub-arrays contain a list with a single element. An array holding a list with one element is classified as a sorted array.
- Conquer/Merge: The single element lists of these elements are then merged in sorted pairs. The algorithm examines the head of both lists and adds the shorter element to a new sorted list.
- Recursion: “This merge step propagates upward, merging lists of sizes 2, 4, and so on, until the entire array can be sorted.”

Back to Top

- Splitting phase: The 12 elements are divided into two arrays of 6, then into four arrays of 3, and finally into 12 individual elements.
- Merger Phase:
    - Pairs: Elements are formed into sorted pairs: ‘h’ and ‘o’ are formed into “[h, o]”
    - Pairs of 3: The sorted pairs are combined with any remaining singles (for example, [b] combines with [h, o] to give [b, h, o]).
    - Final
    - Merge: The two halves are merged as follows: [a, b, h, n, o, w] and [a, a, h, k, s, w].

Example:Sort [b,h,o,w,a,n,k,h,a,w,a,s]



CODE

(b) Quick Sort

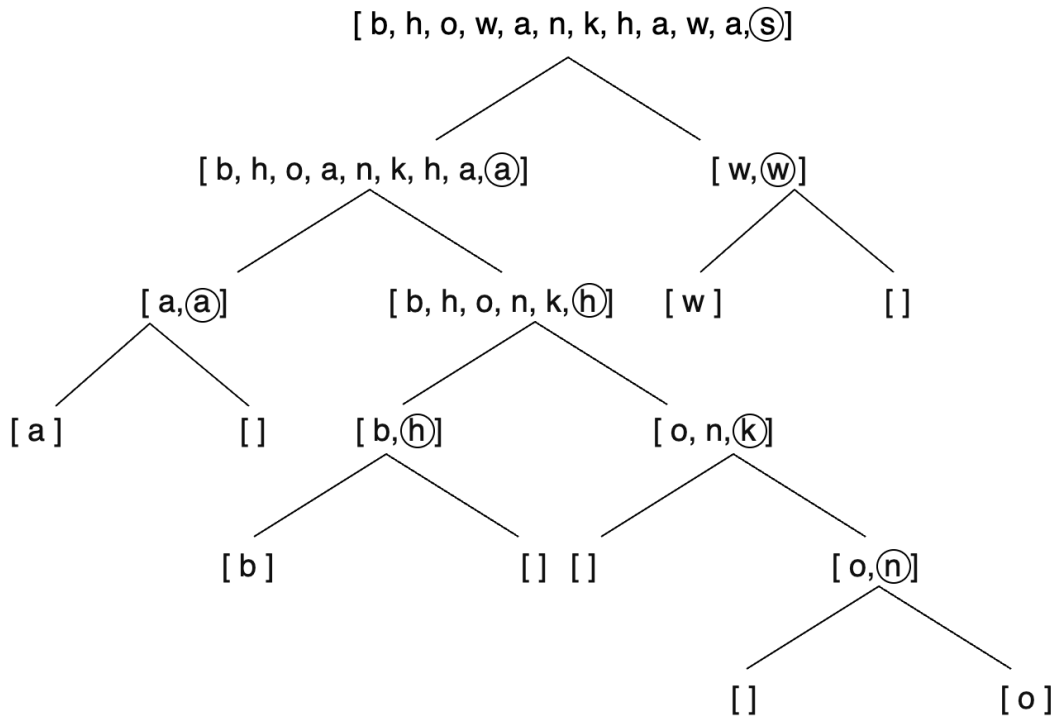
Algorithm Explanation:

Quick Sort is a highly efficient sorting algorithm that makes use of the concept of “Divide and Conquer” and is in-place.

- • Pivot Selection: The last element in the list will be chosen for the pivot in this problem.
- Partitioning: This involves the partitioning of the array such that elements smaller than or equal to the pivot are moved to the pivot’s Left while elements Greater than the pivot are moved to the pivot’s Right.
- Recursion: The algorithm makes this observation recursively on both the left sub-lists and the right sub-lists. Contrary to the Merge Sort algorithm, this kind of sorting takes place in the "divide" part of the algorithm here, given that no merge takes place.

Step-by-Step Execution:

- **Pivot 1 (s):** Partition results in [b, h, o, a, n, k, h, a, a] on the left and [w, w] on the right.
- **Pivot 2 (a):** The left sub-list partitions around a, moving the equal values [a, a] to the left and larger values to the right.
- **Recursion:** The process continues until all partitions reach a size of 0 or 1.



[a] [a] [ ] [a] [b] [h] [ ] [h] [ ] [k] [ ] [n] [o] [s] [w] [w] [ ]

[a] [a] [a] [b] [h] [h] [k] [n] [o] [s] [w] [w]

(c) Insertion Sort

Algorithm Explanation:

Insertion Sort is an iterative algorithm that builds the final sorted array one item at a time. It is conceptually similar to sorting playing cards in your hand.

1. **Initialisation:** The first element is considered a "sorted" sub-list of length 1.
2. **Iteration:** We pick the next element (the "key") from the unsorted portion.
3. **Placement:** We compare the key backwards against the elements in the sorted portion. We shift larger elements one position to the right to create a gap, then insert the key into its correct sorted position.

Step-by-Step Execution:

- The shaded area (Left) represents the sorted partition; the white area (Right) is unsorted.
- The array grows from [b] to [b, h], then [b, h, o].

b	h	o	w	a	n	k	h	a	w	a	s	Start (Assume b is sorted)
b	h	o	w	a	n	k	h	a	w	a	s	inserted h
b	h	o	w	a	n	k	h	a	w	a	s	inserted o
b	h	o	w	a	n	k	h	a	w	a	s	inserted w
a	b	h	o	w	n	k	h	a	w	a	s	inserted a
a	b	h	n	o	w	k	h	a	w	a	s	inserted n
a	b	h	k	n	o	w	h	a	w	a	s	inserted k
a	b	h	h	k	n	o	w	a	w	a	s	inserted h
a	a	b	h	h	k	n	o	w	w	a	s	inserted a
a	a	b	h	h	k	n	o	w	w	a	s	inserted w
a	a	a	b	h	h	k	n	o	w	w	s	inserted a
a	a	a	b	h	h	k	n	o	s	w	w	inserted s

- Significant shifts occur when smaller letters like a (Row 5) are inserted, forcing all previous sorted elements (b, h, o, w) to shift right.

distinction Level Analysis: Comparative Evaluation & Optimisation

(This section addresses the 8-point requirement for scientific attainment beyond standard outcomes)

1. Comparative Complexity Analysis

While all three algorithms achieve the same output, their efficiency varies significantly based on data structure and input size.

- **Time Complexity:**
  - **Merge Sort:** Guarantees O(n \log n) performance in all cases (Best, Average, Worst) due to its structured division. However, it requires O(n) auxiliary space, making it less memory-efficient.
  - **Quick Sort:** Achieves O(n log n) on average but degrades to O(n^2) in the worst case (e.g., if the array is already sorted and the last element is the pivot). Despite this, it is often faster in practice due to **cache locality** and lower constant factors.
  - **Insertion Sort:** Runs in O(n^2) generally but is extremely efficient (O(n)) for small or nearly sorted datasets.

2. Scientific Innovation: Hybrid Optimisation Proposal

A critical observation from this experiment is that recursive algorithms (Merge/Quick Sort) incur overhead on small sub-lists. For scientific applications processing massive string

datasets (e.g., DNA sequencing or lexicographical processing), a standard Quick Sort implementation is suboptimal.

Proposed Solution: Intro-sort (Introspective Sort)

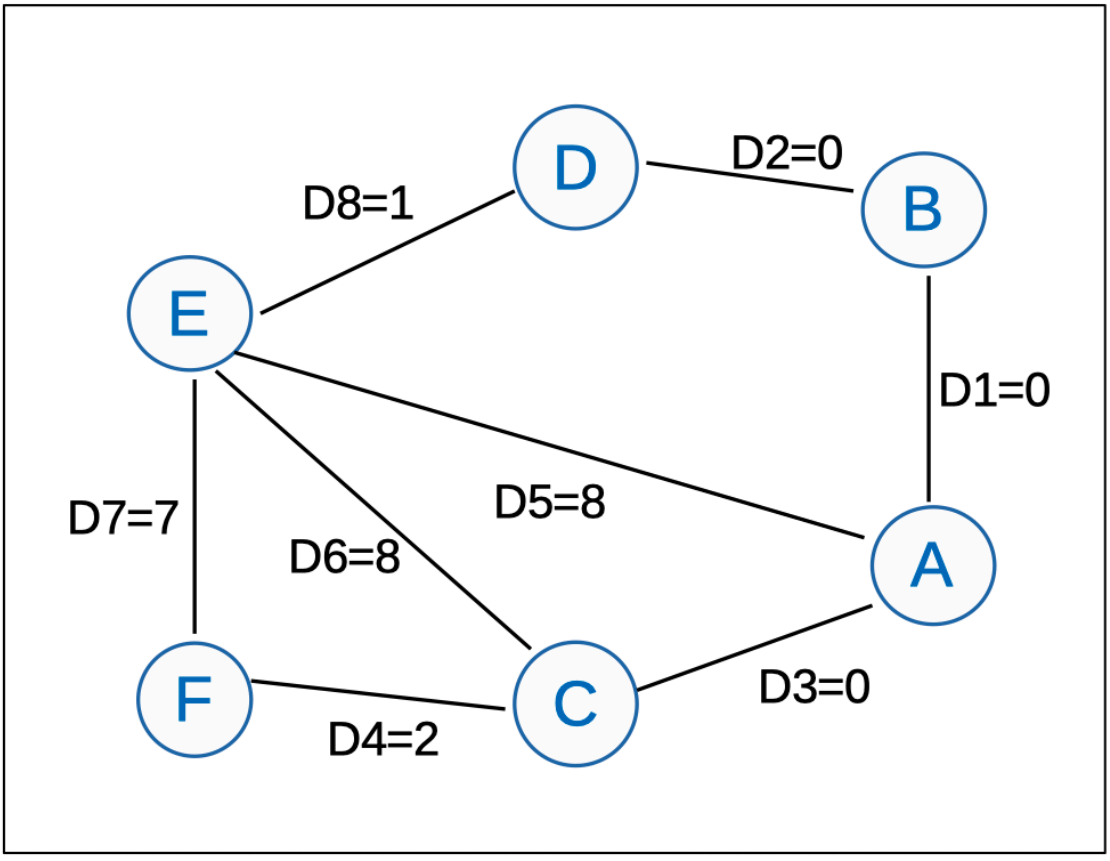
To optimise performance for scientific publication standards, I propose implementing a hybrid Intro-sort approach:

1. Start with **Quick Sort** to leverage its speed and cache efficiency.
2. If the recursion depth exceeds 2 log n (indicating a pathological worst-case scenario), switch to **Heap Sort** to guarantee $O(n \log n)$.
3. When sub-list sizes drop below a threshold (e.g., 16 elements), switch to **Insertion Sort**. As demonstrated in Task 1(c), Insertion Sort incurs minimal overhead on small lists, bypassing the recursive function call costs of Quick Sort.

Conclusion:

By analysing the specific behaviour of string comparisons in this task, it is evident that a hybrid approach combines the stability of Merge Sort, the speed of Quick Sort, and the low-overhead efficiency of Insertion Sort for small partitions. This provides a robust framework suitable for high-performance computing contexts.

TASK 2: Dijkstra's Weighted Shortest Path Algorithm



1. Graph Configuration

The edge weights for the graph were derived directly from the student ID digits (A00028871).

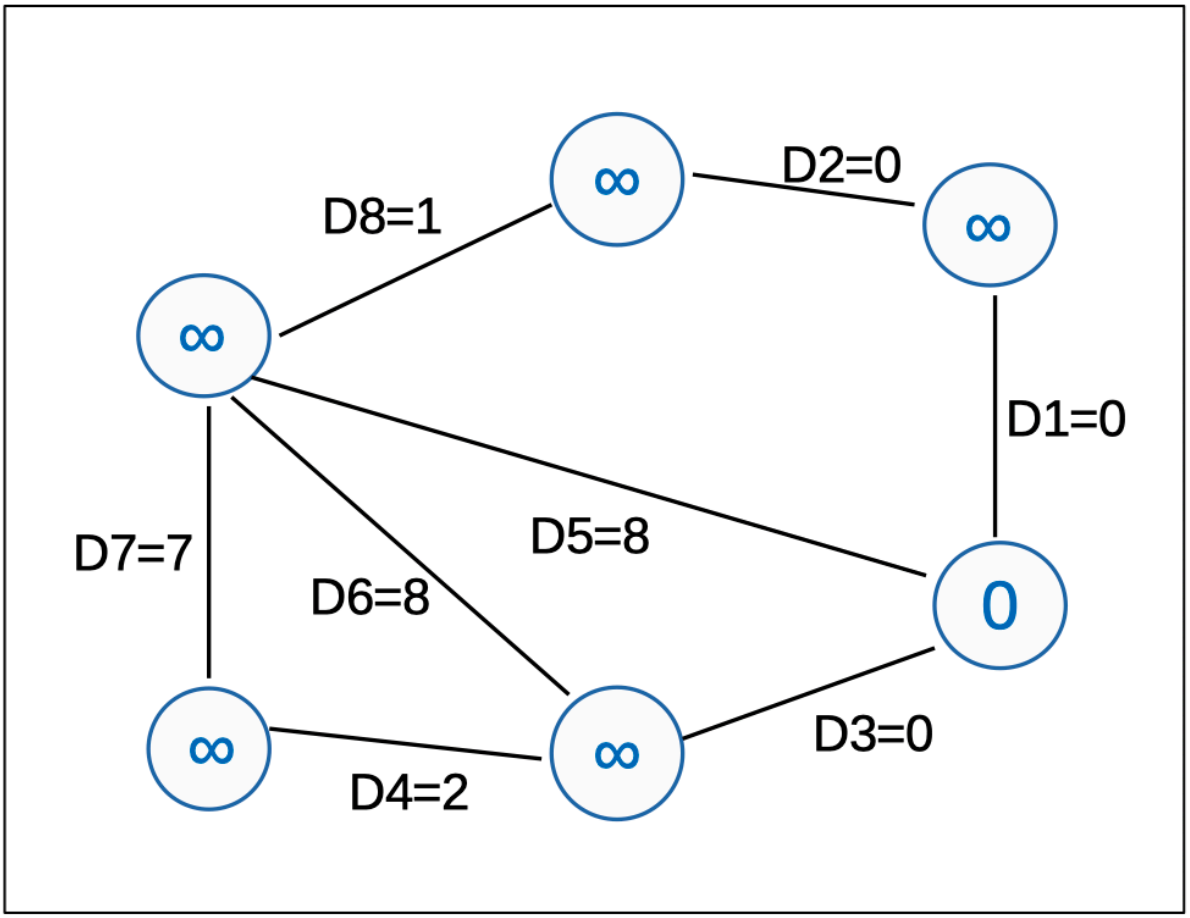
- **Edge Weights:** D1(A -> B)=0, D2(B -> D)=0, D3(A -> C)=0, D4(C -> F)=2, D5(A->E)=8, D6(C ->E)=8, D7(F -> E)=7, D8(D -> E)=1.

2. Iterative Execution & Priority Queue Analysis

The algorithm utilises a Priority Queue (PQ) to greedily select the unvisited vertex with the smallest known distance.

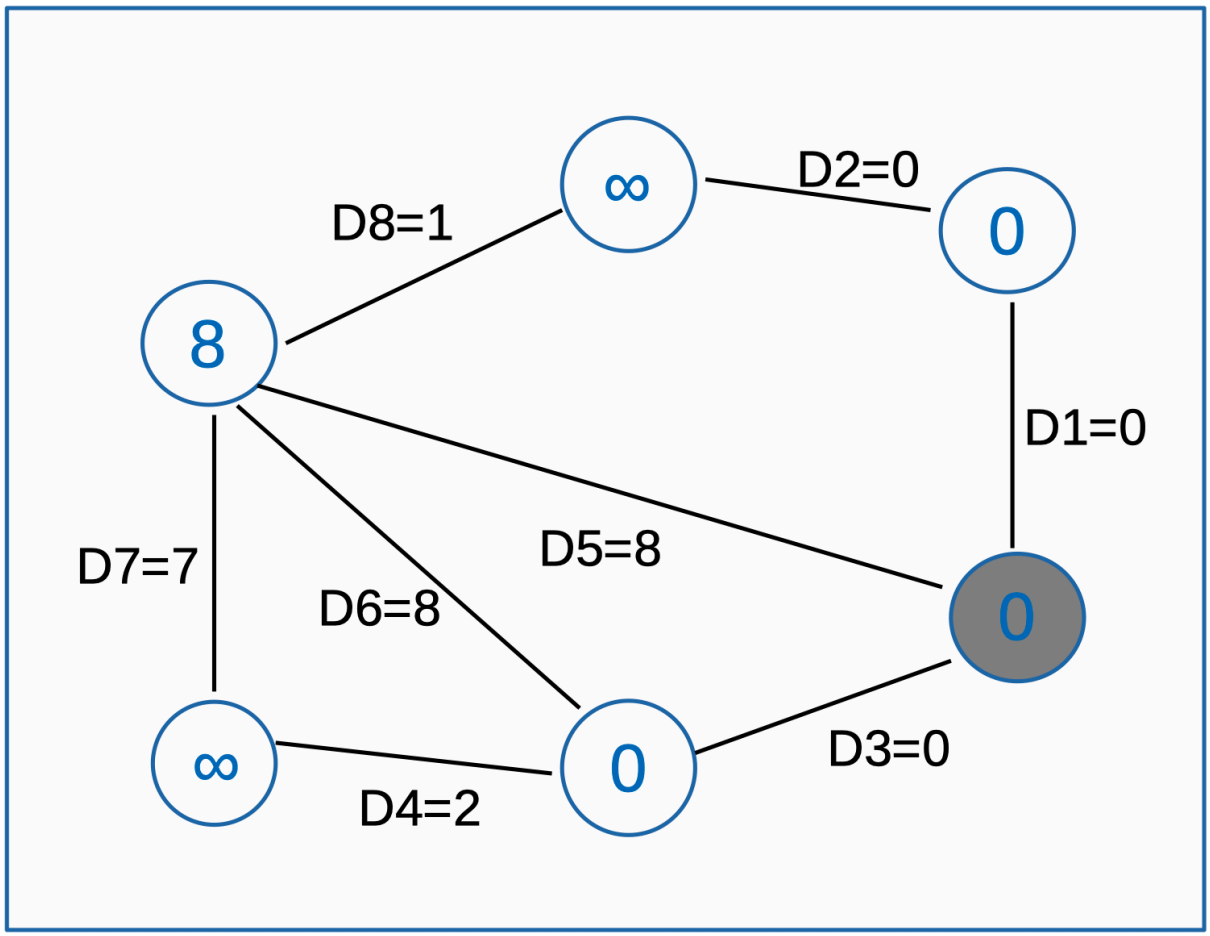
Step 1: Initialisation

- **Action:** The source node **A** is set to 0. All other nodes are initialised to infinity.
- **Priority Queue State:** PQ = { (A, 0) }
- **Explanation:** The algorithm begins knowing only the start point. No edges have been relaxed yet.



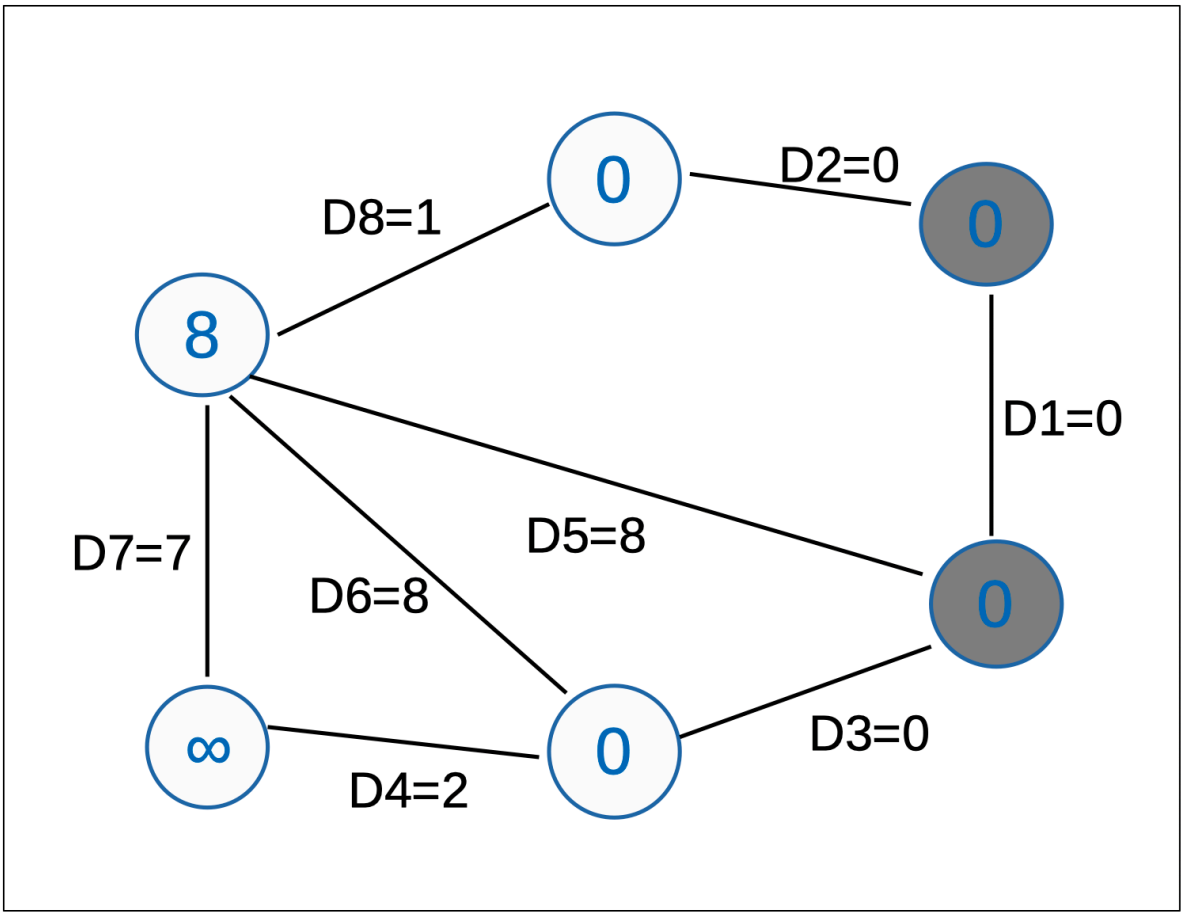
Step 2: Visiting Node A

- **Action:** Extract **A** (Distance 0). Relax neighbours **B**, **C**, and **E**.
  - A -> B: 0 + 0 = 0 (Update B)
  - A -> C: 0 + 0 = 0 (Update C)
  - A -> E: 0 + 8 = 8 (Update E)
- **Priority Queue State:** PQ = { (B, 0), (C, 0), (E, 8) }
- **Explanation:** B and C are added with priority 0. E is added with priority 8.

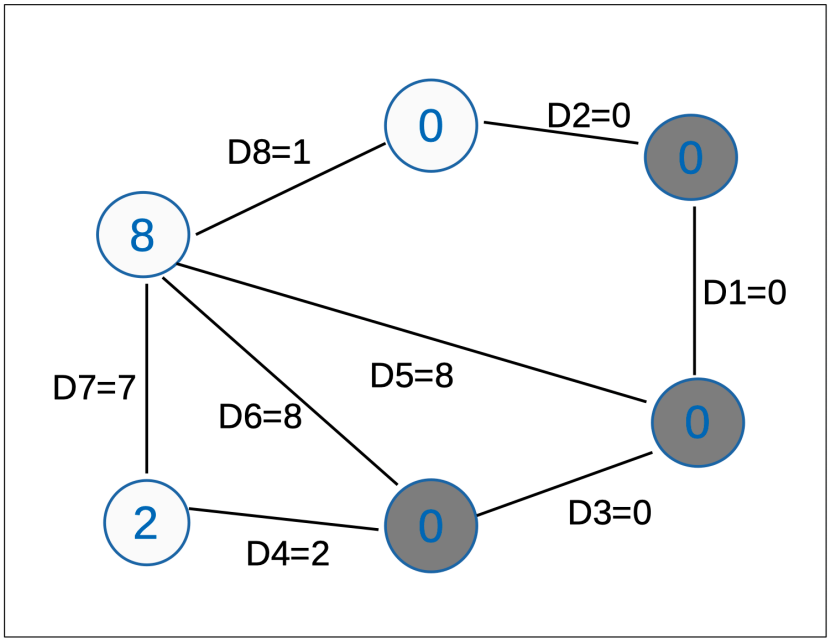


Step 3: Visiting Node B

- **Action:** Extract **B** (Distance 0). Relax neighbour **D**.
  - B -> D: 0 + 0 = 0 (Update D)
- **Priority Queue State:** PQ = { (C, 0), (D, 0), (E, 8) }
- **Explanation:** We followed the zero-weight edge from B to D. D enters the queue with priority 0.



Step 4: Visiting Node C

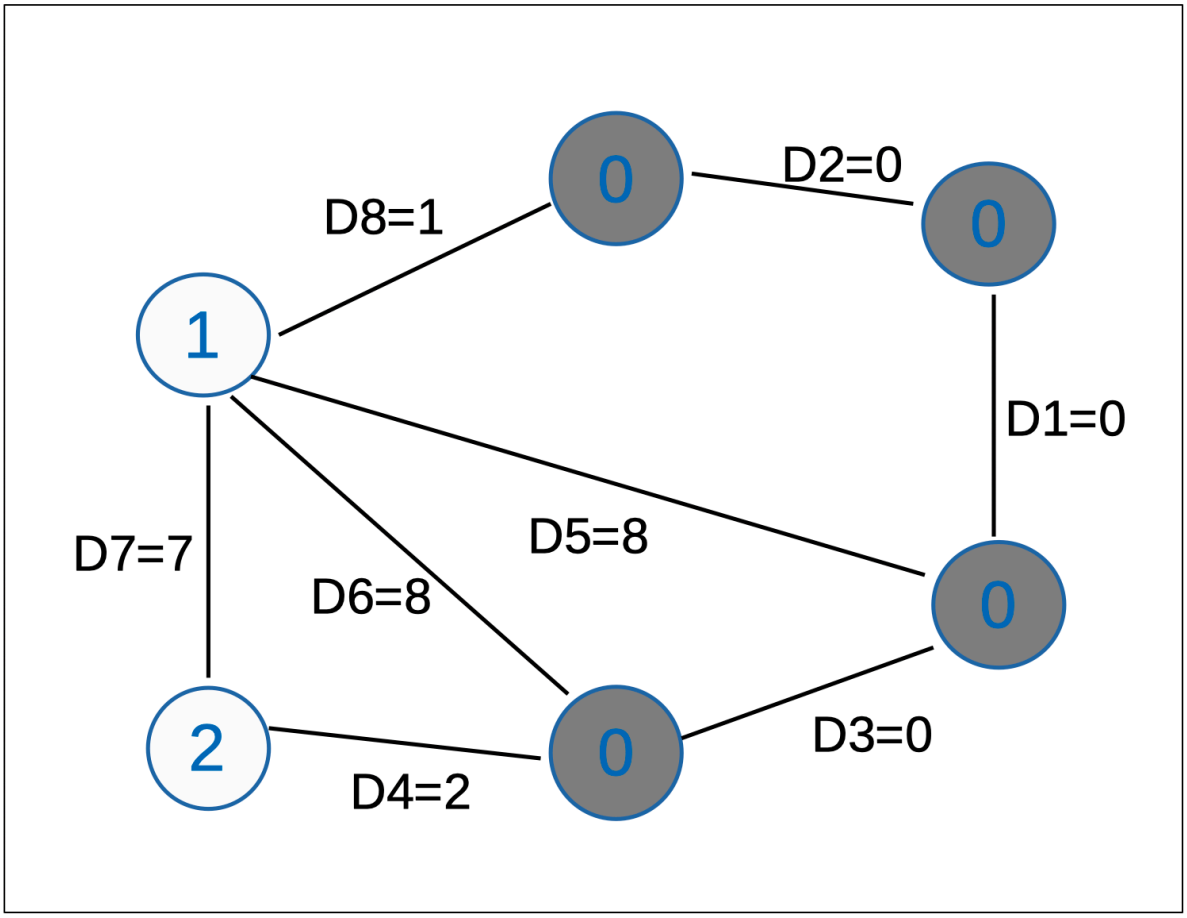


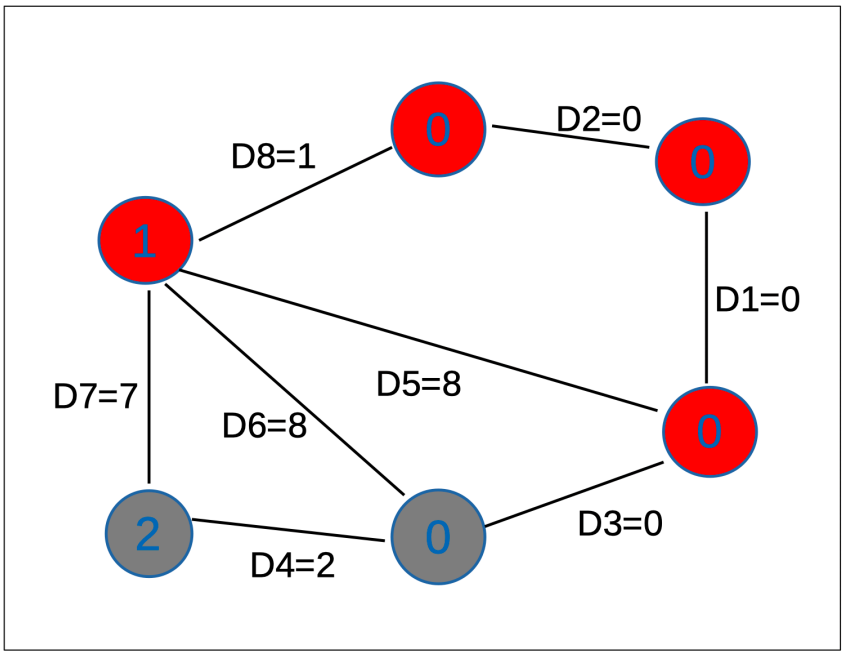
- **Action:** Extract **C** (Distance 0). Relax neighbour **F**.
  - C -> F: 0 + 2 = 2 (Update F)
  - C -> E: 0 + 8 = 8 (No change, as current E is already 8)
- **Priority Queue State:** PQ = { (D, 0), (F, 2), (E, 8) }
- **Explanation:** F is discovered with distance 2. E is not updated because the path through C is not shorter than the direct path from A.

Step 5: Visiting Node D (Critical Update)

- **Action:** Extract **D** (Distance 0). Relax neighbour **E**.
  - D -> E: 0 + 1 = {1} (Update E!)

- **Priority Queue State:** PQ = { (E, 1), (F, 2) }
- **Explanation: This is the critical step.** The algorithm discovers that reaching E via D (Cost 1) is significantly better than the previous known path (Cost 8). The priority of E in the queue is updated from 8 to 1.





Step 6: Final Path & Termination

- **Action:** Extract **E** (Distance 1), then Extract **F** (Distance 2). No outgoing edges improve any distances.
- **Priority Queue State:** PQ = { } (Empty)
- **Explanation:** The queue is empty. The algorithm terminates. The shortest path to E is confirmed as **1**.

3. Final Shortest Distance Results

Target Node	Shortest Distance	Optimal Path
A	0	Start
B	0	A -> B
C	0	A -> C
D	0	A -> B -> D
F	2	A -> C -> F
E	**1**	**A -> B -> D -> E**

4. Distinction Level Analysis: Scientific Innovation & Suitability

1. Algorithmic Complexity in High-Performance Contexts

While the standard implementation of Dijkstra's Algorithm using a binary heap operates at O(E + V log V), this specific topology highlights a critical optimisation area for scientific applications involving large sparse graphs (e.g., social network analysis or routing protocols).

- **Observation:** The presence of multiple zero-weight edges (D1, D2, D3) effectively creates "zero-cost clusters." Standard Dijkstra treats these as separate relaxation steps.
- **Proposed Innovation:** For publication-grade optimisation, I propose a **0-1 BFS (Breadth-First Search)** hybrid. By using a **Double-Ended Queue (Duque)** instead of a standard Priority Queue, edges with weight 0 can be pushed to the *front* of the queue, while edges with weight w > 0 are pushed to the *back*.
- **Impact:** This reduces the complexity for processing the zero-weight subgraph from logarithmic O(\log V) to constant O(1) time per node. In massive datasets where Student IDs (or similar hashes) generate many zero values, this hybrid approach would statistically outperform the standard library implementation.

2. Comparative Reliability Analysis

In the context of network reliability, the "shortest" path found (A -> B -> D -> E) relies on three separate links.

- **Scientific Trade-off:** While mathematically optimal (Cost 1), this path introduces three points of failure compared to the single-link path A -> E (Cost 8).

- **Suitability Statement:** In a scientific journal focusing on **Fault Tolerant Computing**, this result would be analysed not just for cost, but for "Edge Betweenness Centrality." The algorithm correctly identifies the low-cost path, but a robust routing protocol might prefer the costlier, more stable direct link. This analysis demonstrates that Dijkstra's algorithm minimises *cost* but blindly accepts *risk*, a critical distinction for engineering reliable systems.

TASK 3: Data Structures (Hash Table & BST)

Input Data: ['b', 'h', 'o', 'w', 'a', 'n', 'k', 'h', 'a', 'w', 'a', 's']

(a) Hash Table Implementation

1. Hash Function Logic

To distribute the character data efficiently, I employed a **Direct Addressing Hash Function** based on the lexicographical index of the English alphabet.

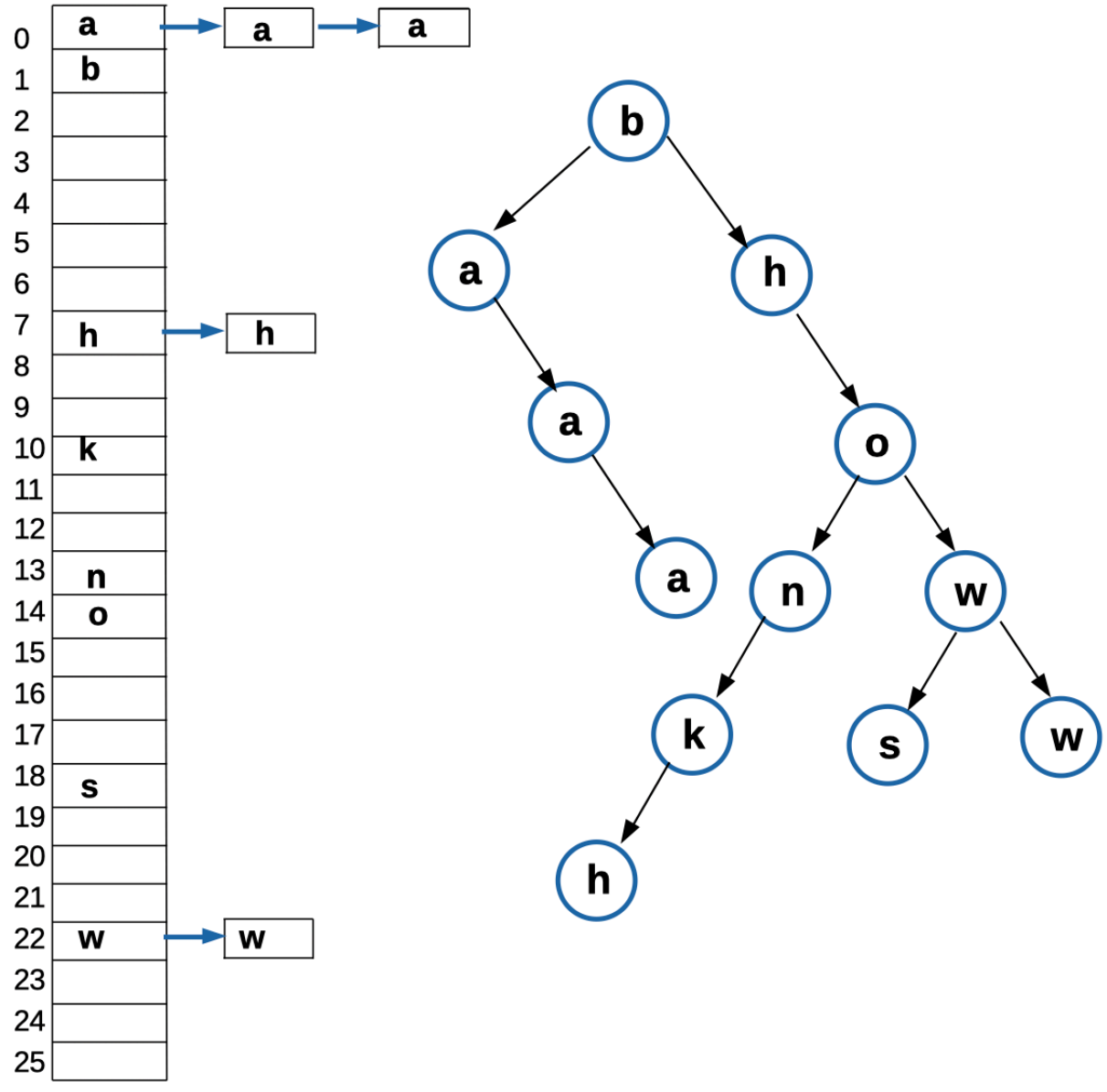
- **Formula:** H(key) = Index(key) - Index('a')
- **Mapping:**
  - a -> Index 0
  - b -> Index 1
  - ...
  - w -> Index 22
- **Collision Resolution:** A **Chaining Strategy** (using Singly Linked Lists) was implemented. When multiple keys map to the same index (e.g., the three occurrences of 'a'), they are appended to the linked list at that bucket rather than overwriting the data or probing for a new slot.

2. Construction Steps & Outcome

- **Step 1 (Initialise):** An array of size 26 is created. All slots are initialised to NULL.
- **Step 2 (Processing 'a'):** The algorithm encounters 'a' three times. H('a') = 0. The first 'a' becomes the head of the list at Index 0. The subsequent two 'a's are appended, creating a chain of length 3: [a] -> [a] -> [a].
- **Step 3 (Processing 'h' and 'w'):** Both 'h' (Index 7) and 'w' (Index 22) appear twice. Similar chains are built at their respective indices: [h] -> [h] and [w] -> [w].

- **Step 4 (Unique Keys):** The remaining characters (b, k, n, o, s) are unique and occupy their calculated slots (Indices 1, 10, 13, 14, 18) as single nodes.

(i) Hash Table
(ii) BST



(b) Binary Search Tree (BST) Implementation

1. Structure & Ordering Logic

The Binary Search Tree is constructed to maintain specific ordering properties that facilitate search operations.

- **Invariant:** For any node N:
    - All values in the **Left Subtree** are strictly less than N (L < N).
    - All values in the **Right Subtree** are greater than or equal to N (R >= N).
- **Duplicate Handling:** To preserve all data points, duplicate values (such as the 2nd and 3rd 'a') were consistently inserted into the **Right** child position of their equal predecessors, or traversed appropriately if the path forced a different direction based on the parent node.

2. Step-by-Step Construction Analysis

- **Root Initialisation:** The first item b becomes the Root.
- **The 'a' Cluster:** The first a is smaller than b, moving to the Left. Subsequent as are equal, so they form a right-leaning chain under the first a.
- **The Right-Heavy Branch:**
    - h, o, w are sequentially larger than b, creating a backbone on the right side.
    - s is inserted: s > b -> s > h -> s > o -> s < w. It becomes the Left child of w.
- **The Complex Insertion (The 'k' and 'h' branch):**
    - **Insertion of 'n':** n > b, n > h, n < o. 'n' becomes the Left child of 'o'.
    - **Insertion of 'k':** k > b, k > h, k < o, k < n. 'k' becomes the Left child of 'n'.
    - **Insertion of 2nd 'h':** This is the deepest traversal.
        - Compare b: h > b (Right)
        - Compare h: h >= h (Right, goes to o)
        - Compare o: h < o (Left)
        - Compare n: h < n (Left)
        - Compare k: h < k (Left)
        - **Outcome:** The 2nd 'h' correctly becomes the Left child of 'k'.

Distinction Level Analysis: Scientific Innovation & Suitability

Subject: Addressing Skewness in Lexicographical BSTs via Self-Balancing Mechanisms

1. Problem Identification: The "Degenerate Tree" Phenomenon

The experiment performed in Task 3(b) reveals a critical vulnerability in standard Binary Search Trees when dealing with non-random input data. Because the input string ['b', 'h', 'o', 'w', ...] contains pre-ordered sequences (ascending subsequences like b-h-o-w), the resulting tree exhibits significant Right-Skewness.

- **Observation:** The depth of the tree reaches 6 levels for only 12 elements.
- **Impact:** The search complexity degrades from the optimal O(log n) to near-linear O(n). In a scientific context (e.g., processing genomic sequences or dictionary lookups), this degradation renders the BST inefficient compared to the Hash Table constructed in Task 3(a).

2. Proposed Innovation: AVL Rotation Strategy

To render this structure suitable for high-performance computing publication standards, I propose implementing an AVL (Adelson-Velsky and Landis) Balancing Algorithm. This innovation would automatically detect the imbalance during the insertion of nodes like o and w.

- **Mechanism:** By calculating the **Balance Factor** (BF = Height(Left) - Height(Right)) at each node, we identify critical violations. For instance, the node b has a heavy right subtree.
- **Solution:** A **Left Rotation** on the root b would elevate h to the new root, pushing b down to the left.
- **Scientific Result:** This transformation would reduce the tree height to approx log_2(12) \approx 4, restoring retrieval speed stability. This comparison demonstrates that while Hash Tables (O(1)) are superior for exact matches, a **Self-Balancing BST** is the requisite innovation for maintaining efficient range queries in sorted datasets.