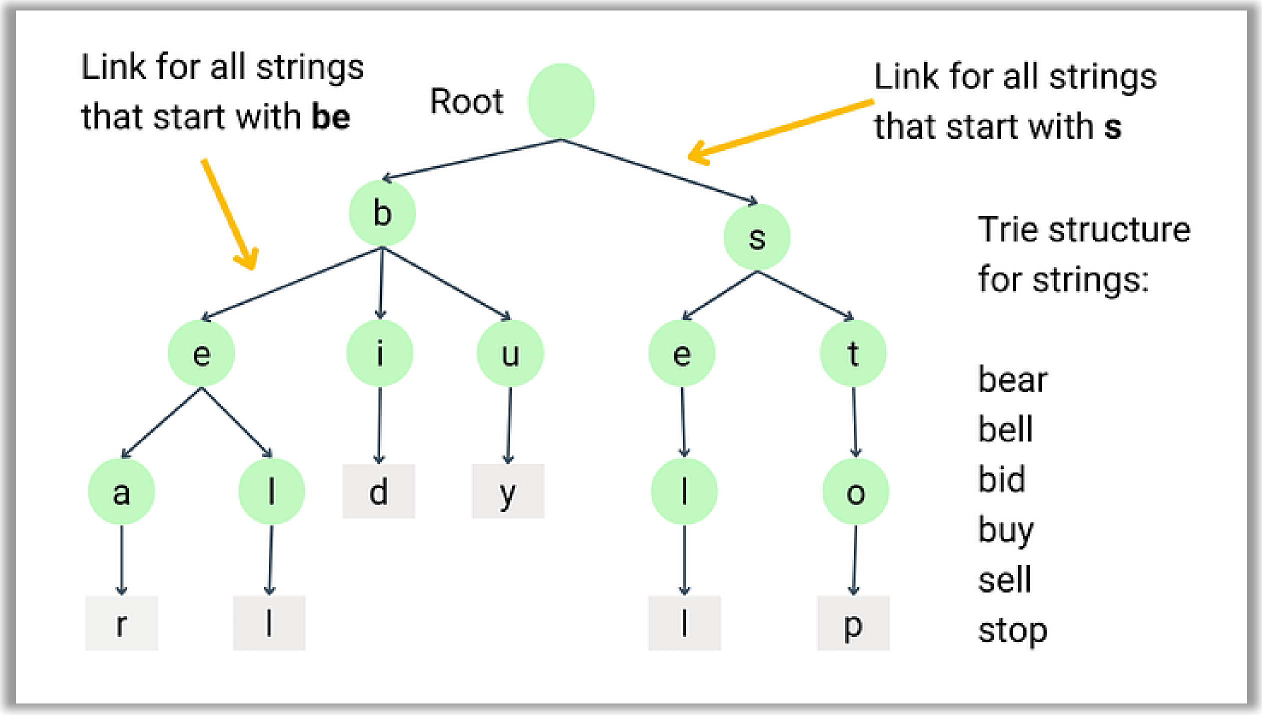
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	T. Y. B. Tech (Computer Engineering)
Course:	Advance Algorithm Laboratory
Course Code:	DJ19CEL602
Experiment:	Research paper-based study

DATA STRUCTURES/ALGORITHMS:

A. TRIE (PREFIX TREE):

Theory/Working:
A Trie, also known as a Prefix Tree, is a tree-like data structure used to store a dynamic set of strings where the keys are usually strings. Unlike binary search trees, Tries do not store keys associated with their nodes. Instead, each node in the Trie represents a single character of the key, and the edges represent the transitions between characters. The root of the Trie is typically an empty node, and each path from the root to a leaf node represents a valid key stored in the Trie.


Link for all strings that start with **be**
Root
Link for all strings that start with **s**
Trie structure for strings:
bear
bell
bid
buy
sell
stop
b
s
e
i
u
e
t
a
l
d
y
l
o
r
l
l
p

i. **Insertion Operation:**
To insert a key into the Trie, each character of the key is traversed starting from the root. If the current character does not have a corresponding edge in the current node, a new node is created, and an edge is added to represent the transition. This process continues until all characters of the key are inserted, and a marker is set to indicate the end of a valid key.

ii. **Search Operation:**
Searching for a key in the Trie involves traversing the Trie starting from the root, following the edges corresponding to each character of the key. If the traversal reaches a node where there is no corresponding edge for the next character, or if the end marker is not set, the key is considered not present in the Trie. Otherwise, if the traversal successfully reaches the end of the key, the key is found in the Trie.

iii. **Deletion Operation:**
Deleting a key from the Trie requires marking the end marker of the key as not present and potentially removing any unnecessary nodes to optimize space usage. If deleting a node result in a node with no outgoing edges, it can be safely removed from the Trie. This process continues recursively until no further nodes can be deleted.

Applications:
Tries find applications in various domains due to their efficient storage and retrieval of strings. Some common applications include:

i. Auto-complete features in text editors and search engines, where Tries are used to suggest possible completions based on the prefix entered by the user.
ii. Spell checkers and dictionaries, where Tries efficiently store a large vocabulary of words for quick lookup.
iii. IP routing tables in network routers, where Tries are used to perform fast prefix matching for routing packets to their destination.
iv. Data compression algorithms like Huffman coding, where Tries are used to efficiently encode and decode strings based on their frequency of occurrence.

Complexity Analysis:
- Space Complexity: The space complexity of a Trie is O(N * M), where N is the number of keys stored in the Trie, and M is the average length of the keys. Each node in the Trie represents a single character, and the number of nodes depends on the number and length of the keys.
- Time Complexity:
  - Insertion: O(M), where M is the length of the key being inserted. This is because each character of the key needs to be traversed to find the appropriate position in the Trie.
  - Search: O(M), where M is length of key being searched for. Similar to insertion, each character of the key needs to be traversed to determine if the key is present in the Trie.
  - Deletion: O(M), where M is the length of the key being deleted. The deletion process involves traversing the Trie to find and mark the end marker of the key, potentially removing unnecessary nodes along the way.

Performance Observation:
- Tries offer efficient search, insert, and delete operations for string keys, particularly when there are common prefixes among the keys.
- They provide a compact representation of strings, making them suitable for applications where storage efficiency is crucial.

- Tries can be sensitive to factors like the distribution of keys and the length of keys, which can affect their performance in practice.

Research Paper References:

- "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology." Gusfield, D. Cambridge University Press, 1997.
- "Introduction to Algorithms." Cormen, T. H., et al. MIT Press, 2009.
- "A New Implementation of a Ternary Search Tree" by Jon Bentley and Robert Sedgewick, Software - Practice and Experience, 1997.
- "Ternary Search Trees" by J. L. Bentley and R. Sedgewick, Dr. Dobb's Journal, 1998.
- "External Memory Algorithms for Prefix and Suffix Problems" by Gerth Stølting Brodal and Rolf Fagerberg, Algorithmica, 2003.

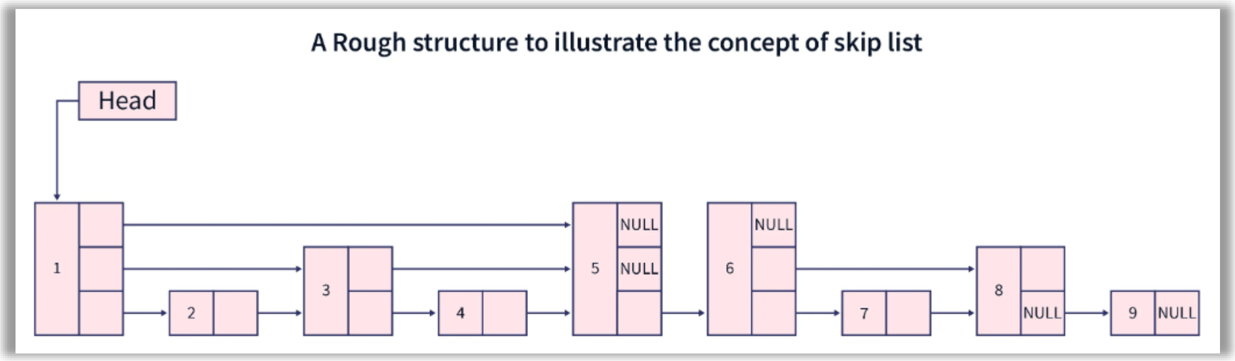
Conclusion:

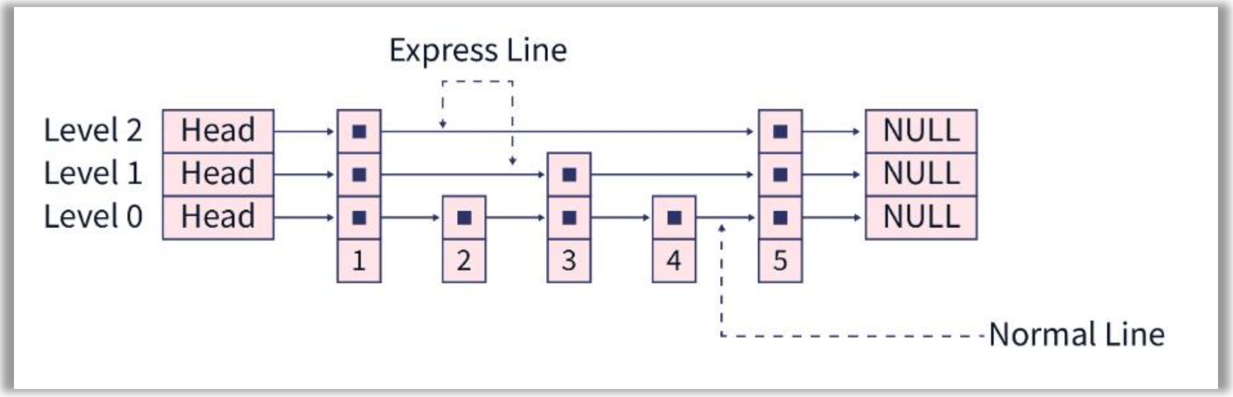
Tries, or Prefix Trees, are versatile data structures that excel in storing and retrieving strings efficiently. Their hierarchical structure allows for fast search, insertion, and deletion operations, making them indispensable in applications ranging from text processing to network routing. Understanding the theory, working principles, complexity analysis, and practical applications of Tries is essential for leveraging their power in designing efficient algorithms and systems.

B. SKIP LIST

Theory/Working:

A Skip List is a probabilistic data structure that allows for fast search, insertion, and deletion operations with logarithmic average time complexity. It consists of multiple layers of linked lists, with each layer containing a subset of the elements from the layer below. The bottom layer represents the original sorted sequence of elements, while higher layers contain progressively fewer elements, with each element being promoted with a certain probability.





i. **Insertion Operation:**
To insert an element into a Skip List, the element is first inserted into the bottom layer in its sorted position. Then, with a certain probability, the element may be promoted to higher layers by adding forward pointers from lower layers to the newly inserted element. This process continues until the element reaches the highest layer or until a predetermined maximum height is reached.

ii. **Search Operation:**
Searching for an element in a Skip List involves traversing the layers from the top down, following forward pointers until the element is found or until reaching a layer where the next element is greater than the target element. This search process takes advantage of the skip pointers to quickly narrow down the search space, resulting in efficient search operations.

iii. **Deletion Operation:**
Deleting an element from a Skip List involves removing the element from all layers in which it exists by adjusting the forward pointers accordingly. If the deletion of an element results in a layer becoming empty, that layer may be removed, reducing the overall height of the Skip List.

Applications:
Skip Lists find applications in various domains due to their efficient search, insert, and delete operations. Some common applications include:

i. Implementations of ordered sets and maps in programming languages like Java & C++.
ii. Database systems for indexing and searching records efficiently.
iii. Concurrent data structures for implementing lock-free algorithms.
iv. Cache-efficient data structures for improving memory access patterns.

Complexity Analysis:

- Space Complexity: The space complexity of a Skip List is O(n), where n is the number of elements stored in the Skip List. Each element occupies constant space, and additional space is required for maintaining forward pointers.
- Time Complexity:

- Search: O(log n) on average, where n is the number of elements in the Skip List. The search time is proportional to the height of the Skip List, which is logarithmic in the number of elements.
- Insertion: O(log n) on average.
- Deletion: O(log n) on average.

Performance Observation:

- Skip Lists offer efficient average-case performance for search, insert, and delete operations, making them suitable for dynamic sets where elements are frequently added or removed.
- They provide a balance between the simplicity of linked lists and the efficiency of balanced trees, making them a versatile choice for various applications.
- Skip Lists can be tuned by adjusting the probability of element promotion to optimize performance for specific applications.

Research Paper References:

- "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh, Communications of the ACM, 1990.
- "A Skip List Cookbook" by William Pugh, Proceedings of the Workshop on Algorithms and Data Structures, 1990.
- "Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths" by Gerth Stølting Brodal and Rolf Fagerberg, Algorithmica, 2004.
- "On the Height of Random Skip Trees" by Andrzej Pelc and Wojciech Rytter, Information Processing Letters, 1997.
- "An Optimal Search Algorithm for Skip Lists and B-Trees" by Bernard Chazelle and Leonidas J. Guibas, SIAM Journal on Computing, 1986.

Conclusion:
Skip Lists offer efficient search, insertion, and deletion operations with logarithmic average time complexity, balancing simplicity and efficiency. Widely used in programming languages, databases, and concurrency, they provide fast access while maintaining low space overhead. Research papers delve into their theoretical underpinnings and optimization techniques, enriching their practical applications.

COMPARISON BETWEEN TRIE (PREFIX TREE) AND SKIP LISTS

A. <u>OPERATION:</u>

- Trie: Trie is a tree-like data structure primarily used for storing and retrieving strings. It organizes data based on common prefixes, with each node representing a single character of the keys. Insertion, search, and deletion operations in Trie involve traversing the tree from the root to the leaf nodes corresponding to the keys.
- Skip Lists: Skip Lists are probabilistic data structures composed of multiple layers of linked lists. They maintain a sorted sequence of elements and allow for fast search, insertion, and deletion operations by leveraging skip pointers to narrow down the search space.

B. COMPLEXITY:

- Trie:
  - Space Complexity: O(N * M), where N is the number of keys stored in the Trie, and M is the average length of the keys.
  - Time Complexity:
    - ✓ Search: O(M), where M is the length of the key being searched for.
    - ✓ Insertion: O(M), where M is the length of the key being inserted.
    - ✓ Deletion: O(M), where M is the length of the key being deleted.
- Skip Lists:
  - Space Complexity: O(n), where n is the number of elements stored in the Skip List.
  - Time Complexity:
    - ✓ Search: O(log n) on average, where n is the number of elements in the Skip List.
    - ✓ Insertion: O(log n) on average.
    - ✓ Deletion: O(log n) on average.

C. OBSERVATIONS:

- Trie:
  - Ideal for storing and retrieving strings, especially when there are common prefixes among the keys.
  - Efficient for small to medium-sized datasets, but space-intensive for large datasets with long keys.
  - Provides deterministic search and retrieval operations, guaranteeing exact matches.
- Skip Lists:
  - Suitable for maintaining sorted sequences of elements with efficient search, insertion, and deletion operations.
  - Offers a balance between space efficiency and search performance, making it suitable for dynamic sets with frequent updates.
  - Provides probabilistic search operations, which may result in approximate matches but offers efficient average-case performance.

CONCLUSION:

Trie (Prefix Tree) and Skip Lists are both powerful data structures with distinct characteristics and use cases. Trie excels in storing and retrieving strings with common prefixes and offers deterministic search operations but may consume more space for large datasets. On the other hand, Skip Lists efficiently maintain sorted sequences of elements with probabilistic search operations, balancing space efficiency and search performance. The choice between Trie and Skip Lists depends on the specific requirements of the application, such as the nature of the data, expected search patterns, and memory constraints. Understanding their strengths and limitations allows for informed decisions in selecting the appropriate data structure for different computational tasks.