Jasveena - 224001588

TASK 4.2P

Exploring the Skip-List data structure

Part -1

1. ***How exactly is the skip-list constructed from layers of linked-lists? What does each of the linked-lists constituting the skip-list store?***
   Ans – A skip list is like a tower of linked lists. Each layer is a linked list, and each higher layer has fewer elements than the one below. These layers act as shortcuts over the lower list, helping to speed up search and other operations.

- ***The bottom layer is a full sorted linked list of all elements.***
- ***Each higher layer "skips over" some elements from below, giving faster access by jumping further ahead.***
- ***Each element in the list may appear in one or more layers, forming a "tower".***

Each layer (linked list) stores pointers to elements. These elements are part of the complete list at the bottom but appear in higher levels depending on their height (randomly determined).

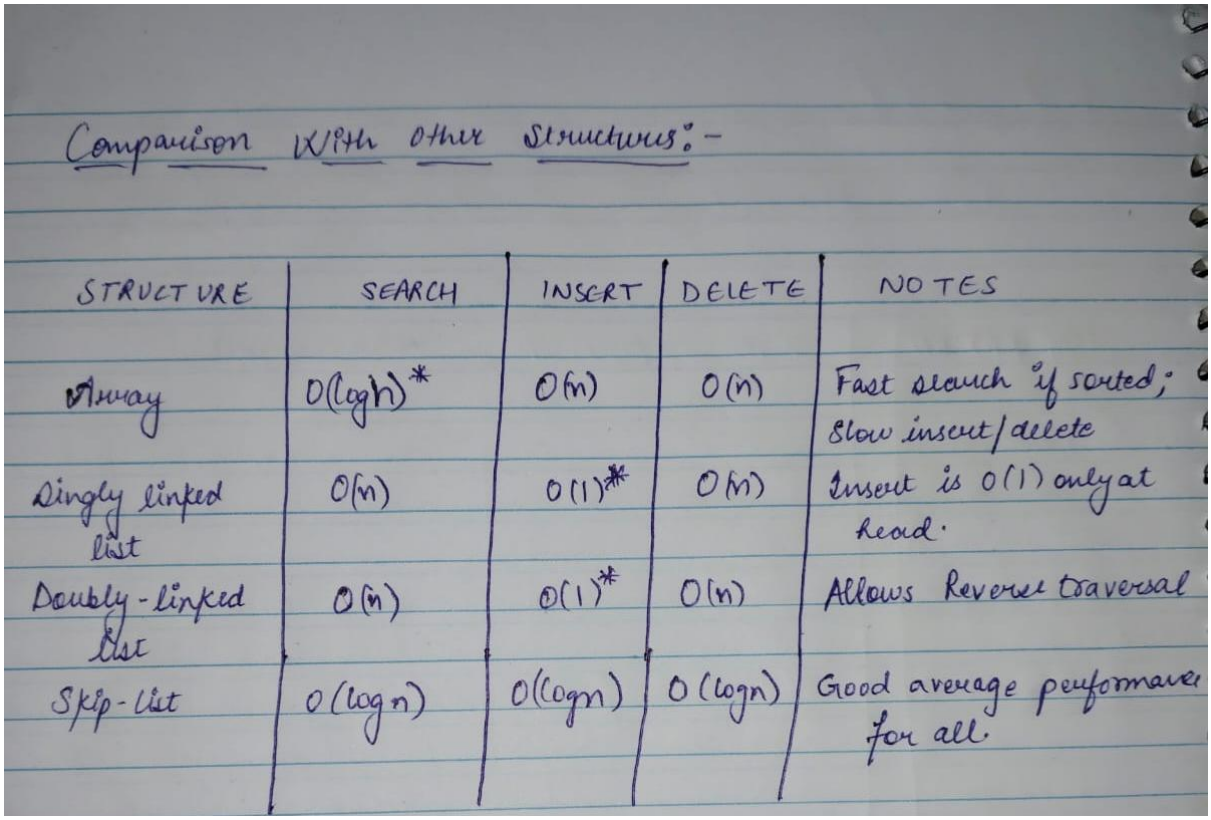
Each node in a layer stores:

- ***The value it represents.***
- ***Pointers: right (next node), and down (node in the layer below).***

2. ***What is the height of a skip-list? How can we determine the height when adding new elements?***
   The height of a skip list is the number of layers it has. To determine the height we can typically use coin tosses, we can :

- ***Start at level 0.***
- ***For each coin toss, if it's Heads, add one more level.***
- ***Stop when we get a Tail.***
- ***The total number of Heads before the first Tail determines the height.***
- ***This random method gives most elements a low height, and a few elements a tall height.***

3. ***What is the runtime complexity for searching, insertion, and removal operations on the skip-list? What does it mean when we say that the complexity of these operations is 'expected'? What is the difference between an expected bound on runtime complexity and a worst-case bound?***



Comparison with other structures:-

STRUCTURE	SEARCH	INSERT	DELETE	NOTES
Array	O(log n)*	O(n)	O(n)	Fast search if sorted; slow insert/delete
Singly linked list	O(n)	O(1)*	O(n)	Insert is O(1) only at head.
Doubly-linked list	O(n)	O(1)*	O(n)	Allows Reverse traversal
Skip-list	O(log n)	O(log n)	O(log n)	Good average performance for all.

- ***The runtime complexity*** ***The runtime complexity of skip-list operations such as searching, insertion, and deletion is O(log n) on average, also known as the expected time complexity. This efficiency is due to the multi-level structure of skip lists, which allows operations to skip over large portions of the list using higher levels.***

  Its like the layers are acting as shortcuts, and we may skip as many elements during our search.

  When we say that the complexity is "expected," we mean that this is the average case performance assuming the random height assignments (usually determined by coin tosses) behave as statistically expected i.e random (o (logn)). However, in the worst-case scenario, where the structure becomes unbalanced (e.g., all elements end up with height 0), the operations could degrade to O(n) time. Therefore, expected complexity reflects typical performance, while worst-case complexity represents the slowest possible case.

-

4. ***Compare the skip-list to other data-structures that you already know, e.g., a list collection, singly and doubly linked lists. What are the advantages and disadvantages of the skip-list? Compare the complexities of the basic operations offered by the skip-list to those of the mentioned data structures.***

Skip lists offer a good balance between simplicity and performance. Their main advantage is that they provide efficient average-case performance for search, insert, and delete operations in O(log n) time, similar to balanced trees, but with simpler algorithms and no need for complex rebalancing. They are also easy to implement and extend dynamically.

However, a disadvantage is that their performance depends on randomness, so in the worst case, operations can degrade to O(n). Compared to arrays (O(log n) search but O(n) insert/delete) and linked lists (O(n) search, O(1) insert/delete at head), skip lists provide a more consistent and balanced approach for dynamic, ordered data.

5. ***How exactly does each of the standard operations work? Here, you should be ready to explain the sequence of actions required to perform searching, insertion, and removal from the skip-list.***
   Search:

- ***Start at the top-left node.***
- ***Move right until the next value is bigger than the target.***
- ***Move down to the next layer.***
- ***Repeat until the bottom layer is reached.***
- ***If found, return the value.***

Insert:

- ***Search for the position as in search.***
- ***Record the path.***
- ***Insert the node at level 0.***
- ***Flip a coin: if heads, go up a level and insert there too.***
- ***Keep flipping until tails (or max height reached).***

Delete:

- ***Search for the node across all layers.***
- ***Remove the node at each level it appears.***

2. first we will define Coin Toss Policy

We'll use this rule:

Tails (T) = stop tower height;
Heads (H) = add one level to the tower (go up one level).
So we keep flipping coins (from left to right) until the first Tail (T) — the number of Heads before it determines the tower height.

- ***If the first toss is T, the tower height is 0 (only bottom level).***
- ***If it's H, H, T, the height is 2 (levels 0, 1, and 2).***
- ***And so on.***

The given tosses are as follows :

T H T T H H T T H T H T H H T H H T T H H T H T T H T T T H T H T H T T H T H H T T

And the values to insert are as follows :

1, 40, 11, 85, 86, 5, 0, 8

Insert 1

First toss is T → height = 0

Level 0: 1

Insert 40

H, then T → height = 1

Level 1: 40

Level 0: 1 → 40

Insert 11

First toss is T → height = 0

Level 1: 40

Level 0: 1 → 11 → 40

Insert 85

First toss is T → height = 0

Level 1: 40

Level 0: 1 → 11 → 40 → 85

Insert 86

Two heads before a tail → height = 2

Level 2: 86

Level 1: 40 → 86

Level 0: 1 → 11 → 40 → 85 → 86

Insert 5

First toss is T → height = 0

Level 2: 86

Level 1: 40 → 86

Level 0: 1 → 5 → 11 → 40 → 85 → 86

Insert 0

First toss is T → height = 0

Level 2: 86

Level 1: 40 → 86

Level 0: 0 → 1 → 5 → 11 → 40 → 85 → 86

Insert 8

One head before a tail → height = 1

Level 2: 86

Level 1: 8 → 40 → 86

Level 0: 0 → 1 → 5 → 8 → 11 → 40 → 85 → 86

Final list

Level 2: 86

Level 1: 8 → 40 → 86

Level 0: 0 → 1 → 5 → 8 → 11 → 40 → 85 → 86