SKIP LISTS:

DRAWBACK OF LINKED LISTS:

- * The worst case time complexity for a linked list is O(n).
- * We have to linearly traverse the linked list to find an element in worst case scenario.
- * Also, it is not possible to jump to the middle or skip elements while searching
- * To overcome these drawbacks, we use skip lists:

SKIP LISTS (Definition)

- * It is a probabilistic Data structure. (an extension of sorted linked list).
- * Subsequent layers are built on top of one another such that each layer has fewer elements and no new elements.
- * The base layer links all nodes in the list.

known as Normal lane

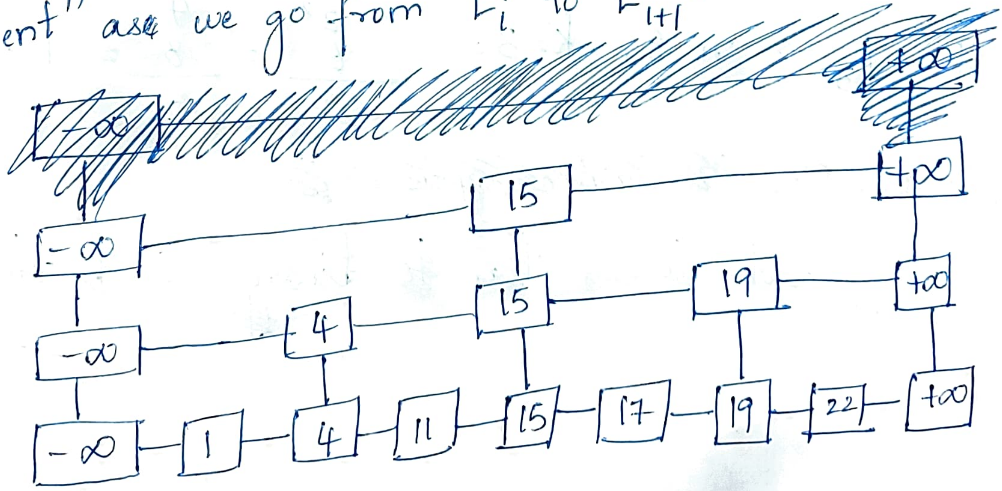
* Skip lists uses −∞ and −∞ keys at each level.

higher layers → Express lane.

TWO APPROACHES TO CONSTRUCT SKIP LIST:

NAIVE APPROACH (DETERMINISTIC)

* In a deterministic skip list, we keep every alternate element" as we go from L_i to L_{i+1}.



PERFECT SKIP LIST

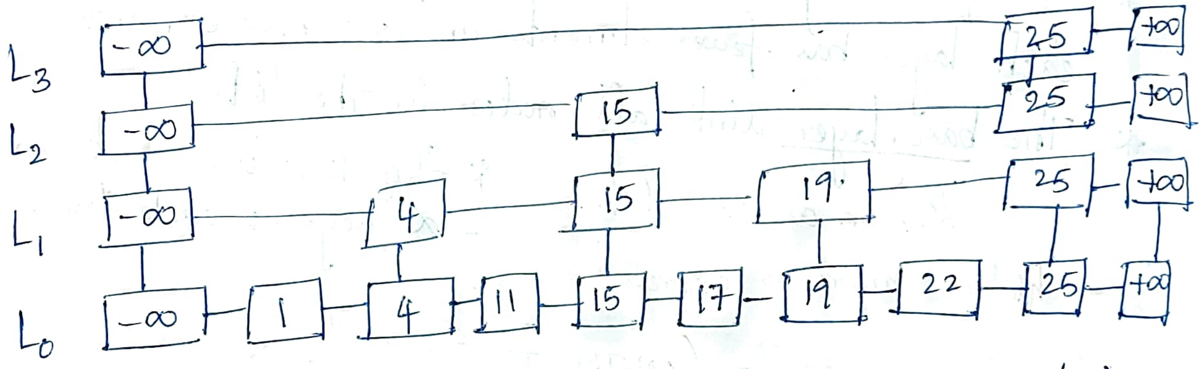
* It is a deterministic skip list with number of nodes in the sorted linked list as even.

* The~~re~~ no. of elements linked at each level is given by:

$$\boxed{\text{No. of elements linked at level } k = \frac{n}{2^k}}$$

n → Total no. of elements in sorted linked list

k → Level no.



At L_0 → $n = 8$, $k = 0$ ⇒ $\frac{8}{2^0} = \frac{8}{1} = 8$. 8 links.

At L_1 → $n = 8$, $k = 1$ ⇒ $\frac{8}{2^1} = \frac{8}{2} = 4$. 4 links.

At L_2 → $n = 8$, $k = 2$ ⇒ $\frac{8}{2^2} = \frac{8}{4} = 2$. 2 links.

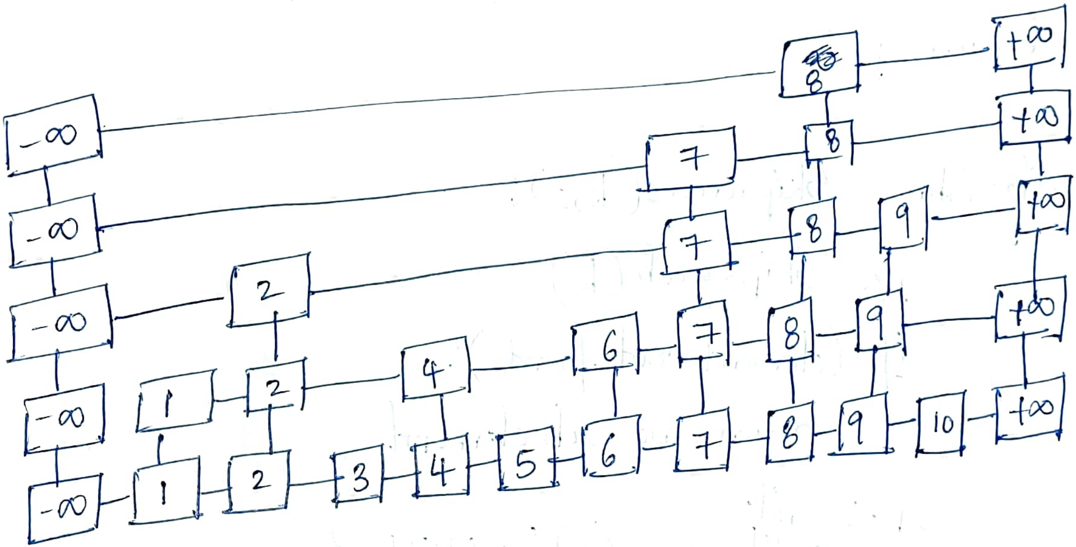
At L_3 → $n = 8$, $k = 3 = \frac{8}{2^3} = \frac{8}{8} = 1$. 1 link.

~~Why we don't use naive approach~~

~~Naive approach is similar to binary search tree~~

RANDOMIZED SKIP LISTS:

- The level-wise links in a randomized skip lists are created using a coin-flip() function.
- When coin-flip() yields '1', we create a link at that level for a given node.
- When coin_flip() yields '0', we don't create a link at that level for a given node.
- We use coinflip() for each node.
- The probability for each outcome of coin-flip() is 1/2.



ALGORITHMS:

```
Algorithm SkipInsert(k,e):
Input : Item (k,e)
Output : None
p ← SkipSearch(k)
q ← insertAfterAbove(p, null, (k,e)) // At bottom level
while random() = 1 do
    while above(p) = null do
        p ← before(p) // scan backward
    p ← above(p) // jump to higher level.
    q ← insertAfterAbove(p,q,(k,e)) //insert new item.
```

```
Algorithm SkipSearch(k):
Input: search key k
Output: Node in S whose item has largest key less
        than or equal to k.
Let p be the topmost, left node of S
while below(p) ≠ null do
        p ← below(p) //drop down
        while key(after(p)) ≤ k do
            Let p ← after(p) //scan forward
return p
```

```
Algorithm Skipremove(k,e):
        p ← Skipsearch(k)
        if (element not found):
                return no such element
        else: while
              ~~until~~ above(p) != NULL do
                 ~~delete (p)~~ link before(p) to after(p)
                 ~~link before (p) to after(p)~~ delete(p)
                 p ← ~~below(p)~~ ~~above~~ ~~p ← below(p)~~ p ← below(p)
              end while
        end Algorithm
```

NOTE: after(p) → Return node following p on same level

before(p) → " " preceding p on " "

below(p) → " " below p in same tower

above(p) → " " above " " " "

COMPLEXITY:

Search, insert, remove → $O(\underbrace{n}_{\text{no. of items}}+\underbrace{h}_{\text{height}})$

Space → $O(n)$

```
Algorithm Skipremove(k,e):
        p ← Skipsearch(k)
        if (element not found):
                return no such element
        else:
            while above(p)!=NULL do:
                    link before(p) to after(p)
                    delete(p)
                    p ← below(p)
            end while
end.
```