Exercise 1 (Skip List). A skip list is a light-weight data structure that allows to insert, delete and search for keys. Additionally, it can answer efficiently to predecessor queries for any key \( k \), that is for the element \( y \) with the largest \( \text{key}[y] \leq k \) stored in the skip list.

A skip list \( S \) for a set \( S \) consists of a number of sorted linked lists \( L_0, L_1, \ldots, L_h \). Each list \( L_i \) stores a subset \( S_i \subseteq S \), such that \( S_0 = S \) and \( S_i \subseteq S_{i-1} \) for all \( 0 < i \leq h \), and \( S_h = \emptyset \). Each sorted list also stores two dummy elements, one with key \( -\infty \) at the beginning of the list and one with key \( +\infty \) at the end of the list. For a set \( S_i \) (or list \( L_i \)) we call \( i \) the level of that set (or list), and we call \( h \) the height of the skip list. We also have pointers between consecutive lists. More precisely, for every element \( x \in S_i \) (with \( i > 0 \)) we have a pointer from its occurrence in \( L_i \) to its occurrence at the lower level in \( L_{i-1} \)— see Fig. 1 for an example.

We denote the pointer from an element \( x \in L_i \) to the next element in \( L_i \) by \( \text{next}[x] \), and the pointer from \( x \) in \( L_i \) to the copy of \( x \) in \( L_{i-1} \) by \( \text{down}[x] \). Answering a predecessor query for key \( k \) works as follows: Start from the dummy element \( -\infty \) at the top level and Repeat until you cannot go down anymore: Go down and Move to the next element at the current level as long as its key is at most \( k \); when we cannot go down anymore, Output the element at which we stopped.

\[ \text{Predecessor}(S, k) \]

| \( S \) | \( k \) | \( y \) | \( \text{key}[y] \leq k \) | \( S \) |
In order to answer queries efficiently, we need at the same time the height to be low, $O(\log n)$, and the elements on a given level to be well spread with respect to elements at the level just below. It would be very costly to maintain such a structure efficiently deterministically. The key to the efficiency of skip list is to use randomness in order to decide at which level to insert a new element. Here is how the insertion of a new element proceeds. First, choose the level $i$ at which $x$ is inserted: for this purpose, toss a fair coin repeatedly until $\text{Tail}$ is obtained and set $i$ to the number of times $\text{Head}$ occurred before $\text{Tail}$ was obtained. Then, $x$ is inserted at every level from $\mathcal{L}_i$ to $\mathcal{L}_0$.

**Question 1.2**

Insert($\mathcal{S}, x$)  

next  
down  

$\mathcal{S}$

Let $n$ denote the total number of elements ever inserted in the skip list. The insertion procedure ensures that every level contains about half the elements in the level just below on expectation. This will guarantee with high probability that the height of the data structure is at most $O(\log n)$ and that the number of elements scanned at each level is $O(1)$ for each query. Let us denote by $\text{height}(x)$ the height at which element $x$ is inserted.

**Question 1.3**

$\Pr\{\text{height}(x) \geq s\} = 1/2^s$  

$s \in \mathbb{N}$

Recall the union bound which states that for any set of events $A_1, \ldots, A_n$ (interdependent or not), $\Pr\{A_1 \lor A_2 \lor \cdots \lor A_n\} \leq \Pr A_1 + \Pr A_2 + \cdots + \Pr A_n$.

**Question 1.4**

$t > 1$  

$1 + t \log_2 n$  

$1/n^{t-1}$  

$h$

We are now ready to prove a bound on the query time in a skip list. Let $X_i$ denote the random variable for the number of next pointers followed in $\mathcal{L}_i$ when answering the query.

**Question 1.5**

$E[X_i] \leq 1$  

$i$

$1/2$  

$\mathcal{L}_i$  

$\mathcal{L}_{i-1}$  

$\mathcal{L}_i$

**Question 1.6**

$O(\log n)$

$3 \log n$

Let us now consider deletion.

**Question 1.7**

Delete($\mathcal{S}, x$)  

$\mathcal{S}$

**Question 1.8**

$O(\log m)$

Let $m$ be the number of elements presently in the skip list ($m \leq n$ and $m$ might be much lower than $n$ if many deletions did occur).