

# 1 Skip List

A skip list (discussed in lectures 11, 12) is a data structure that uses linked lists and randomization to guarantee  $O(\log n)$  time for insert, delete, and search operations on expectation. Work your way through this assignment to learn how to implement a skip list and use it to solve an interesting problem!

## 1.1 Implementation

In this part, your task is to implement a skip list data-structure using a class `SkipList` defined in **SkipList.java**. The `SkipList` class will comprise of the following components:

- **Instance variables:**

1. `int height` - The height of the skip list, equals the maximum number of linked lists any node is part of. An empty skip-list will have height 1.
2. `SkipListNode head` - The head sentinel, has fixed value `Integer.MIN_VALUE`, and height equal to the skip list height. Upon initialization, it should have height 1, and should point to the tail sentinel at that height.
3. `SkipListNode tail` - The tail sentinel, has fixed value `Integer.MAX_VALUE`, and height equal to the skip list height. At each height, it points to null. Upon initialization, it has height 1.
4. `Randomizer randomizer` - A randomizer object to generate node heights for elements to be inserted. See below for more details.

- **Member functions:**

1. `public boolean search(int target)` - Takes an integer *target* as a parameter and returns *true* if it is present in the skip list, and *false* otherwise.
2. `public boolean delete(int num)` - Takes an integer *num* as a parameter and returns *false* if it is not present in the skip list, otherwise removes the **first** occurrence of value *num* from the skip list and returns *true*.
3. `public void insert(int num)` - Takes an integer *num* as a parameter and inserts it into the skip list (**before** all existing instance of *num*, if any).
4. `public Integer upperBound(int target)` - Takes an integer *target* as a parameter and returns the next larger integer in the skip list. Returns `Integer.MAX_VALUE` if no such integer exists.
5. `public void print()` - The **print** function simply prints the elements of skip list.

Each node in a `SkipList` will be an object of the class `SkipListNode`. The class `SkipListNode` comprises of the following:

- **Instance variables:**

1. `int height` - The height of the node, equals the number of linked lists it is part of.
2. `int value` - The value stored at the node.
3. `ArrayList<SkipListNode> next` - An ArrayList of size *height*, with pointers to next nodes of this node. Note that `next[0]` points to the next element in the base linked list (that is, list at height 1), whereas `next[h]` points to the next node in the linked list at height  $h+1$ .

Your task is to implement the functions **search**, **insert**, **delete** and **upperBound** in **SkipList.java**. You can create helper functions if you like, but you are not allowed to change the signature of the given functions. The implementations of the classes **SkipListNode** and **Randomizer**, and the functions **print** and **SkipList** have been provided. You are not allowed to change them.

**Note on Randomizer and Insertion:** An object *randomizer* of class Randomizer has been initialized in the SkipList class. It has a function **binaryRandomGen**, which returns *true* or *false* in a (pseudo) random manner. While inserting a node, you should initialize its height to 1, and using `randomizer.binaryRandomGen()`, you should iteratively check whether its height should be increased. **Note that if the node height becomes larger than the skip list height, then you must stop calls to `binaryRandomGen()` function, and update the skip list height.** Using this procedure is mandatory, not adhering may cause inconsistency in output.

Example:

TestCase :-

```
SkipList skiplistobj = new SkipList();
```

```
skiplistobj.insert(3);
```

```
skiplistobj.insert(1);
```

```
skiplistobj.insert(5);
```

```
skiplistobj.insert(8);
```

```
skiplistobj.search(1);
```

```
skiplistobj.search(10);
```

```
skiplistobj.delete(5);
```

```
skiplistobj.insert(4)
```

```
skiplistobj.search(5);
```

```
skiplistobj.search(4);
```

```
skiplistobj.upperBound(3);
```

Outputs:-

```
true
```

```
false
```

```
false
```

```
true
```

```
4
```

Final SkipList after all operations:-

```
head->1---->4---->tail
```

```
head->1->3->4->8->tail
```

## 1.2 Using SkipLists

In this part, you will learn a beautiful application of the *SkipList* class you created.

Suppose you run a *Bakery* that specializes in multi-tier cakes. The standard preparation procedure is to bake each tier in a separate oven and combine them **bottom up**. Obviously, each subsequent tier should be **smaller** than the one just below. You rose to fame overnight when your special New Year cake gained widespread acclaim on *Instagram*. As a result, you decided to apply your data structure skills to ground the preparation procedure. You have a large number of ovens. You bake one cake tier in each one of them. Cake tiers in different ovens may have different sizes, and they come out of the oven one by one, randomly. As and when a cake tier comes out, you have two choices -

1. You can make it base of a new cake.
2. You can add it on top of an existing tier (whose top most cake has larger size).

You **must** process the cake levels in the order they come out of the oven. Assume that no two cakes come out at the same time. You want to make the **minimum** number of cakes, owing to your limited space availability.

**Your task is to implement the function:**

`static Integer solve(ArrayList<Integer> cakes)` in **Bakery.java**. The function takes an ArrayList of Integers representing the sizes of the cake tiers. It returns the minimum number of cakes that can be formed. You can create helper functions if you like, but you are not allowed to change the signature of the given function.

This question can be solved with various data structures. However, for an optimal solution, you require  $O(\log n)$  time for search, insert and delete. You will be using a skip list here, as it provides the required time guarantees. You should use the functions developed in the first part of this question. **You are not allowed to use any other inbuilt/user defined data structure for this question.**

Example:

Input Format:-

The input will consist of an ArrayList containing the sizes of the cake tiers in the order they come out of ovens.

Output Format:-

The output will consist of a single integer which represents the minimum number of cakes that the baker can make.

TestCase-:

Input:

2 9 3 2 8

Output-:

3

Explanation-:

An optimal way will follow the following procedure:

1. Get cake tier sized 2. Create a new base and get ((2)).
2. Get cake tier sized 9. Create a new base and get ((2), (9)).
3. Get cake tier sized 3. Create a new base, and get ((2), (9), (3)).
4. Get cake tier sized 2. Put on 3, and get ((2), (9), (3, 2)).
5. Get cake tier sized 8. Put on 9, and get ((2), (9, 8), (3, 2)).

In the end, we have 3 cakes. It can be shown that this is the minimum possible.

**Hint:** This problem can be solved in an online fashion - you don't need to know what tiers you will be encountering afterward to make the best decision at a given step. Ponder on the following questions:

- What should you store in the skip list? *After seeing some tiers and arranging them in some way, what is the important information you need for later?*
- Is it better to create a new base or add it on top of an existing one?
- Once you choose to insert the tier on an existing one, which one should you choose to insert if there are several options?