

Lab 3

*Due Date: March 18**No Late Penalty Through: March 22nd, 4:00pm*

CAREFULLY READ THIS ENTIRE ASSIGNMENT BEFORE BEGINNING!

Basic Assignment:

In this lab you will implement the skip list data structure. While in reality, this would then be used by some application, in this lab no such application will be provided (or written by you). Your goal is hence to implement and debug your skip list class and provide output to demonstrate to us that you have achieved this task.

For the purposes of this lab, we are going to make the following simplifications to the real problem. First of all, your implementation will not save a `DataRecord` that is associated with each key. However, just as you did in your `CS241Dictionary` you could add a field to each item that would be a reference of a `DataRecord` (which would be a clone of the `DataRecord` provided with `insert`) which could then be returned as desired for the other operations. Secondly, you need not check in `insert` if the keys were duplicates. Depending on the application duplicate keys may be acceptable. Finally, for the purpose of demonstrating that your skip list class is working properly you are to directly print the specified information directly from within your `Skiplist` class. I recommend that you use a method that only prints in “debugging mode”. Then you can have a constructor function in which you specify that you want the skiplist to be in debugging mode.

Of course, you will need a simple driver that makes calls to your `Skiplist` class. You can make this as simple as you like (e.g. it could directly make the desired calls to the skiplist methods). Nothing is being provided except for a procedure to simulate a coin flip. Also, you may want to use the `Terminal` class provided with Lab 1 (which also prints everything to the terminal when a transcript is being made) or Lab 2 (which does not print anything to the terminal when a transcript is being made).

In order to earn a “B” you must correctly implement the following methods. For your implementation to be correct it’s not sufficient that it have the correct behavior (of course this is a necessary condition for being correct). Here’s a concrete example of a common error I’ve seen in the past. Suppose that you are inserting a new item and its size (as selected by the random process described in class) is 1 (so it only will be inserted in L_0). I’ve seen many students then search for the item’s position in list L_0 and then insert it. While the skiplist will have the proper structure, this has expected time complexity of $O(n)$ which makes this very inefficient. Thus someone making this mistake should expect a deduction! After all, this course is about designing EFFICIENT algorithms and data structures. It does no good to design an algorithm for the `insert` method with expected time complexity $O(\log n)$ and then implement it in a way that has expected time complexity $O(n)$.

Here are the methods that are required for the basic assignment.

`print` takes no parameters and has no return value. You are expected to use this method (when in debugging mode) after every call to a method that changes the skiplist. This should not be a public method, but rather it should be private and called at the end of `insert` and `remove` when in debugging mode. Here's an example of the expected output:

```
Here is the current skiplist:
```

```
Level 3: 241
```

```
Level 2: 241
```

```
Level 1: 15 25 102 241 333 727
```

```
Level 0: 15 25 87 102 241 333 441 630 727 827 925
```

`insert` takes one parameters, the key (an integer) and has no return value.

For the basic assignment you can enforce the maximum size for an item to be 8 (levels 0, ..., 7). In the unlikely event that you randomly select a size bigger than 8, just use 8 instead. While you can allocate head and tail to initially have size 8, it would be good to keep track of the maximum level used so that `print`, `search`, `insert`, etc. can begin there.

For demonstrating that `insert` is working correctly you MUST provide a list of every comparison made at each level. Here's example output from inserting 600 into the skiplist shown above:

```
Inserting 600
```

```
At level 3 compared 600 to: 241  infinity
```

```
At level 2 compared 600 to: infinity
```

```
At level 1 compared 600 to: 333  727
```

```
At level 0 compared 600 to: 441  630
```

```
Here is the current skiplist:
```

```
Level 3: 241
```

```
Level 2: 241
```

```
Level 1: 15 25 102 241 333 727
```

```
Level 0: 15 25 87 102 241 333 441 600 630 727 827 925
```

`search` takes one parameters, the key (an integer) and returns a boolean that is `true` if the item was found and `false` otherwise.

For demonstrating that `search` is working correctly you MUST provide a list of every comparison made at each level. Here's example output from searching for 400 in the skiplist shown above:

```
Searching for 400
```

```
At level 3 compared 400 to: 241  infinity
```

```
At level 2 compared 400 to: infinity
```

```
At level 1 compared 400 to: 333  727
```

```
At level 0 compared 400 to: 441
```

```
It was not found.
```

`predecessor` takes one parameter, the key (an integer) and returns an integer which is the predecessor (with special values NOTFOUND if the key is not in use and NONE if there is no predecessor since the key was the minimum).

Here's a sample output:

```
Searching for 827 to find its predecessor.
At level 3 compared 827 to: 241  infinity
At level 2 compared 827 to: infinity
At level 1 compared 827 to: 333  727  infinity
At level 0 compared 827 to: 827
The predecessor of 827 is 727
```

If it is not found or has no predecessor an appropriate message should be printed.

`successor` takes one parameter, the key (an integer) and returns an integer which is the successor (with special values NOTFOUND if the key is not in use and NONE if there is no successor since the key was the maximum.) The output here should look similar to that shown above for `predecessor`.

`minimum` takes no parameter and returns an integer which is the minimum key (with a special value NONE if the set is empty). Your output here need just be a single line indicating the minimum value (or that there is none if the set is empty)

`maximum` takes no parameter and returns an integer which is the maximum key (with a special value NONE if the set is empty). Your output here need just be a single line indicating the maximum value (or that there is none if the set is empty)

What to Submit

You should submit all code that your wrote along with two transcripts from different runs of your driver (with the same set of calls). The calls made by your driver should demonstrate to us that your methods are all correct. Points can be deducted if you did not do a good job in testing your methods (as demonstrated by the provided output). It is adequate to just insert 10 items. YOUR TRANSCRIPTS SHOULD NOT EXCEED 5 PAGES EACH.

Additional Feature:

- (1/3 letter grade) Implement a remove procedure which takes a single parameter, the key and returns a boolean that is `true` if the key was found (and hence deleted) and `false` if the key was not found. Your output should be similar to that from `insert` along with some indication of the return value.
- (2/3 of a letter grade) Do not make any assumptions about the maximum size. You must begin with head and tail having size 2 (so there are only two levels, level 0 and level 1). Then if during insert you randomly select the size of a node to be larger than your current size of the head and tail, then you should increase the size of your head and tail to the smallest power of 2 that is adequate. Whenever this happens the debugging

output from `insert` should indicate so. Don't be surprised to find that you don't need to do this very often. Notice if you increased the size of head and tail by just one element then you would be okay (on average) until the number of items doubled in value. Thus by doubling the size of head and tail you don't expect to do this again until the number of elements is n^2 where n was the number of elements when you doubled the size of head and tail.

You must also keep track of the largest level which was ever selected so that all searches can begin at that level. Otherwise, `insert`, `search`, `remove`, etc. would be less efficient because they would be spending time at levels in which there were never any keys. While this would still result in methods with $O(\log n)$ expected time complexity it would noticeably increase the leading constant and you do want to do this!