

Lab 3

October 14, 1999

DUE DATE: Tuesday November 9

Carefully read the assignment. If you don't understand anything, ask Dr. Goldman or one of the TAs about it.

“B” Portion:

For this lab you will implement the skiplist data structure. We are going to make the following simplifications. First, you will not maintain a `DataRecord` for each key. However, just as you did in your `CS241Dictionary` you could easily add a field to each item that would be a reference of a `DataRecord`. Secondly, you need not check in `insert` if the keys were duplicates. Depending on the application duplicate keys may be acceptable. So that you can check/demonstrate that your `SkipList` class is working properly, your `SkipList` constructor should take a boolean (`debugMode`). On the web page under Lab 3, you will find some useful provided code (and code fragments). Included in this are two simple drivers (`driver1` for the “B” portion and `driver2` for those who implement `remove`) that make calls to your `SkipList` class which will be used for testing. Also, a procedure (to be incorporated as part of your `SkipList` class) to simulate a coin flip is being provided. If you are using Java the provided drivers are set up so that you can give a command line argument for the transcript file. Or if you prefer you can uncomment out the marked line so that a transcript file `transcript.txt` is always made. If you are using C++, some help on overloading the `<<` operator is with the provided code.

In order to earn a “B” you must correctly implement the following methods. For your implementation to be correct you must properly implement each of the `SkipList` methods. 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 using only list L_0 and then insert it in the appropriate place. While the skiplist will have the proper structure, this error causes `insert` to have an expected time complexity of $O(n)$ which makes this very inefficient. Thus someone making this mistake should expect a deduction. 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.

`toString` (or in C++ overload the `<<` operator) which takes no parameters and returns a string (or `ostream` in C++). When in debugging mode, the expected return value should be something like:

```
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
```

When not in debugging mode, the expected return value should be something like:

```
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 “B” portion 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, use 8 instead. While you can allocate head and tail to have size 8, it would be good to keep track of the maximum level used so that print, search, insert, etc. can begin there.

When in debugging mode (so that it can be confirmed that insert is working correctly) you MUST provide a list of every comparison made at each level. Here’s example output from within `insert` when inserting 600 into the skiplist shown on page 1:

```
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
```

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

In debugging mode, you MUST provide a list of every comparison made at each level. Here’s example output from within `search` when searching for 400 in the skiplist on page 1. It should return `false`.

```
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
```

`predecessor` takes one parameter, the key (an integer) and returns an integer which is the predecessor (with special values `NOTFOUND = -1` if the key is not in use and `NONE = -2` if there is no predecessor since the key was the minimum).

Here’s a sample output (when in debug mode) for finding the predecessor of 827. It should return the value 727.

```
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
```

`successor` takes one parameter, the key (an integer) and returns an integer which is the successor (with special values `NOTFOUND = -1` if the key is not in use and `NONE = -2` 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 the special value `NONE = -2` if the set is empty).

`maximum` takes no parameter and returns an integer which is the maximum key (with a special value `NONE = -2` if the set is empty).

“A” Portion:

- (1/3 letter grade) Implement a `remove` method 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. When in debug mode your output should be similar to that for that in `search`.

- (2/3 of a letter grade) Do not make any assumptions about the maximum size of an item. 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. (To do this you must reallocate a new, larger, head and tail, copy over the needed information, initialize the rest, and make any other changes required in your skiplist.) Whenever head and tail are resized (and you are in debug mode) you should print something like:

```
resizing --- increasing head and tail size from 2 to 4
```

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.

To receive full credit here, 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

What to Submit

You should submit the code you wrote for your `SkipList` class (and any other code YOU wrote) along with your output using the *provided driver1* (if you did NOT implement `remove`) or *driver 2* (if you implemented `remove`).

Notice that you should get different output each run (since, as shown with the provided code, the random number generator is suppose to be seeded with the clock and hence different each time). Since there are 13 items inserted you expect that at least one of them has size 3 or more (i.e. is in levels 0, 1, 2 and perhaps more). If in your run this does not occur, please run the driver again and provide us output from a run in which at least one item has size 3 or more. This should naturally happen after only a few runs.

Please do NOT submit any of the provided code. (This includes the drivers.)