Skipping Through Lists

Motivation

In applications like Google Search, it's necessary to map words that appear in web pages with the URLs that the web pages appear on. For example, the words "research" and "funding" appear on the WWU website with the URLs https://foundation.wwu.edu/ and https://gradschool.wwu.edu/admissions. We need a data structure that supports the fast addition and retrieval of words with associated pages.

One possible solution is a hash table, as realized by java.util.HashMap. Hash tables can work very well in $O(1)$ time to map a word string to a value (the set of URLs). However, this is true only when the hash table is very sparse and there are few collisions between words and their associated hash values. Consider the English language and how many words there are (not to mention other languages that your search engine must support). In order to make the hash table approach work well, the array storage would have to be many times the size of the English language. You'd need a computer with a gigantic amount of RAM in order to make this work.

Linked lists may solve the storage problem (you only store the words and their pages), but searching through a linked list is very slow at *O(n)*. What is a computer scientist to do?

Skip Lists

Skip lists are a better approach for this. A skip list is a probabilistic data structure that allows $O(log\ n)$ insertion and $O(log\ n)$ retrieval, with storage that is comparable to a link list.

Remember that the $\log_2$ of a million is about twenty -- that means that the growth in complexity is really very manageable when you deal with lots of words.

A skip list is a linked list that is built in layers.

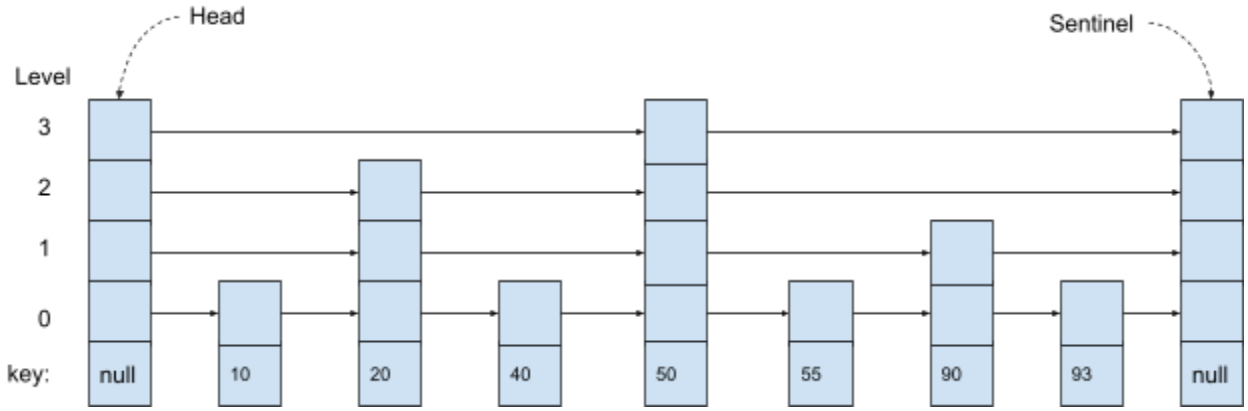


Figure 1. A skip list with seven elements.

In the figure above, we see a skip list with seven elements. This is a four level skip list. At level 0 is the standard linked list. All elements are linked together in nodes. There is a special node at the front called the Head, and a special node at the rear called the Sentinel.

Higher levels act as "express lanes" for the levels below. An element at layer i appears at level $i + 1$ with probability p, where p is usually $\frac{1}{2}$.

Looking up an Element

A search for an element starts at the highest level (level 3 in the figure above). It proceeds horizontally until the current element is greater than or equal to the search target, or the Sentinel is reached. If the current element is equal to the target, then the target has been found. Otherwise, we start the search again from the previous element by dropping down vertically one level.

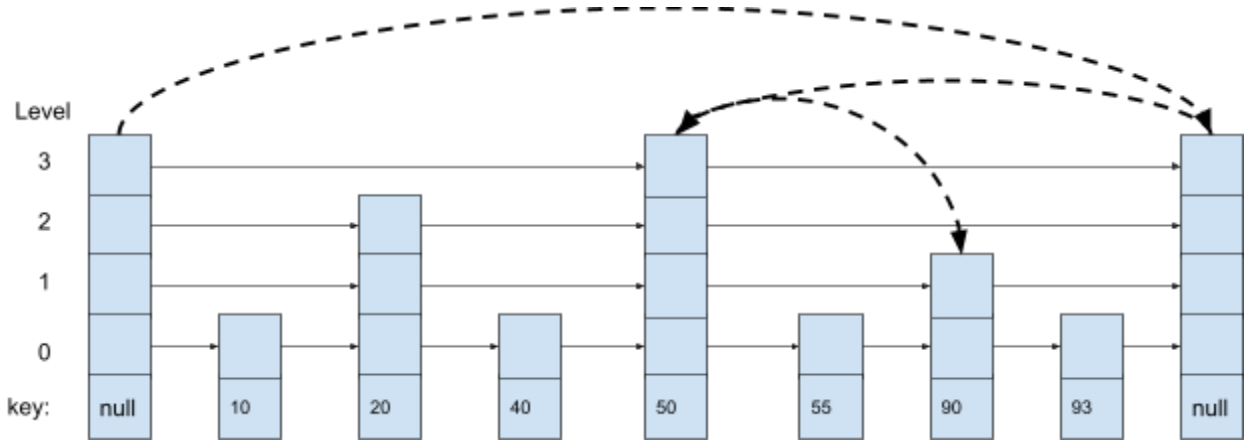
Example 1

Let's use the example in Figure 1. Suppose we are searching for the element with the key 90. The search begins at level 3 of the Head. It proceeds horizontally at level 3 to the node containing the key 50. Since 50 is less than 90, we continue horizontally to the next element.

The next node is the Sentinel, so we go back to the node with key 50. We drop down one level to level 2. We then proceed horizontally to the next node.

The next node is again the Sentinel, so we go back to the node with key 50. We drop down one level to layer 1. We proceed horizontally to the next node.

The next node at level 1 is the node with key 90, so we have found the target. The figure below illustrates this sequence.



Example 2

Let's use the example in Figure 1. Suppose we are searching for the element with the key 40. The search begins at level 3 of the Head. It proceeds horizontally at level 3 to the node containing the key 50. Since this is more than 40, we go back to the Head, and drop down to level 2.

From level 2, we go horizontally to the node with key 20. Since this is less than 40, we proceed horizontally to the node with key 50. Since 50 is more than 40, we go back to the node with key 20. We drop down to level 1 at the same node with key 20. The next horizontal node at level 1 is again the node with key 50. So we go back to the node with value 20, and drop down to layer 0.

At the node with key 20, we proceed horizontally. The next node has the key 40, and hence we have found the target. The figure below illustrates the sequence in this example.

Inserting an Element

The insertion algorithm is similar to the lookup algorithm. First use the lookup algorithm to find the correct place to insert the node. Then insert the node at layer 0.

After that, go to the next layer up. Flip a coin, and if it comes up heads, insert the node at that higher layer. If it comes up tails, then you are done. Continue in this manner until you flip tails, or you have reached the top layer.

Performance of Skip Lists

It should be apparent that the "express lanes" in the higher layers enable you to skip past lots of elements. On average, you will be able to skip past ½ the elements at each level, since every element present at level *i* has a probability of ½ that it appears in level *i* + 1. This property of being able to eliminate ½ the elements at each step is what gives skip lists an $O(lg\ n)$ growth rate. We will see this property again and study it in more detail when we study binary trees.

Your Tasks

Your task is to implement a skip list in **SkipList.java**.

Task 1

Your first task is to implement the Node class. This generic class is used to hold a key and a value. It also holds an array of pointers to the next Node at various levels of the list. Level 0 is the lowest level. Higher levels are sparser and sparser. A node with a key of "__HEAD__" and a value of null is the Head. A node with a key of "__SENTINEL__" and a value of null is the tail.

Nodes are parameterized on a generic type V, representing the type of the value. Keys must implement Comparable, so that they can be sorted. For example, Strings all implement Comparable and have the **compareTo** method.

Here are the methods you need to implement for **class Node<V>**. A skip list node doesn't just have one next node pointer. It should have an array of pointers to the next node, one for each level of the skip list.

- **`public Node(Comparable key, V value, int levels)`** - create a Node with the given key, the given value and the given number of levels.
- **`public Comparable getKey()`** - return the key
- **`public V getValue()`** - return the value
- **`public int getLevels()`** - return the number of levels
- **`public Node<V> getNext(int level)`** - return the next node at the given level
- **`public void setNext(int level, Node<V> next)`** - set the next node at the given level

Task 2

Your next task is to implement the class SkipList<V>. Like Node, this class is parameterized on the generic type V for the value type. Keys need to be Comparable, as usual.

Here are the methods you need to implement in the class.

- **`public SkipList(int levels)`** - create a SkipList with the given number of levels
- **`public int getLevels()`** - return the number of levels in the skip list
- **`public Node<V> getHead()`** - return the Head node. A node with the string key `"__HEAD__"` and value null is the Head node.
- **`public Node<V> getSentinel()`** - return the Sentinel node. A node with the string key `"__SENTINEL__"` and value null is the Sentinel node.
- **`public V lookup(Comparable key)`** - lookup and return the value associated with the key or null if not found
- **`public void insert(Comparable key, V value)`** - insert key and value in the skip list.

Your SkipList should use the **`CoinFlipper`** class. A true flip from the coin flipper means that a level should be added, while a false flip means to stop inserting. Please follow this convention to make sure that the autograder checks correctly.

Building and Testing Your Code

You can check if your code compiles with

```
make
```

There is a small piece of test code in a main method in SkipList.java. Feel free to add to it as you write your code. You can run it with

```
make run
```

Automatic tests are in the Test directory. You can run the automatic tests with

```
make test
```