HashPipe

Wed Apr 11 11:25:11 2018 +0200, rev. c7e1d31

Implement a symbol table using a *Hash Pipe* data structure. This exercise introduces yet another implementation of symbol tables (besides those presented in chapter 3 of [SW]) with logarithmic insertion and search. The main difficulty are low-level implementation choices and careful programming similar to what is needed for implemeting a binary search tree of hash table.

Description

Write the class HashPipe that implements many of the methods for an ordered symbol table with String keys and Integer values. In particular, implement the following API:

```
public class HashPipe

public HashPipe() // create an empty symbol table
public int size() // return the number of elements
public void put(String key, Integer val) // put key-value pair into the table
public Integer get(String key) // value associated with key
public String floor(String key) // largest key less than or equal to key
```

Data structure

The *Hash Pipe* data structure is a pointer-based structure that represents every key–value pair using a so-called *pipe*. The Pipes are arranged from left to right, in sorted key order. The entries of each pipe point to the pipes to its right, as shown here:

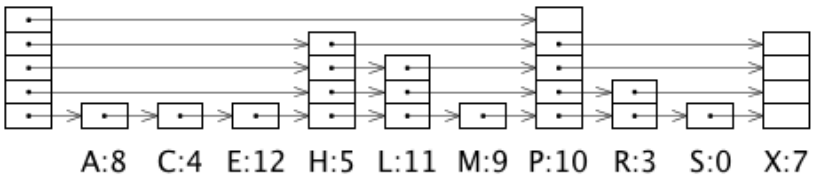

Figure 1: The data structure after running the basic symbol table client in [SW, 3.1] on input `S E A R C H E X A M P L E`.

To be quite precise, consider key k. The pipe's reference at height h points to the smallest key k' larger than k of height at least h. There is a special *root* pipe to the left, which you can think of as 'negative infinity.' The root pipe does not represent any key–value pair, no other pipe points to it, and it is at least as high as any other pipe. References with 'nothing to the right' contain *null*, shown as empty boxes in the figure.

The number of entries (the 'height' of a pipe) is given by the Java hash code of the key,[1] more specifically it is the *number of trailing zeros of* `key.hashCode()` $+1$. For instance, the `"A".hashCode()` is 65, which is 1000001 in binary and has no trailing zeros; thus, the height of the pipe of key "A" is $0 + 1 = 1$. Similarly, the `"P".hashCode()` is 80, which is 1010000 in binary and has four trailing zeros; thus, the height of the pipe of key "P" is $4 + 1 = 5$.

Navigation in a Hash Pipe works very much like navigation in a search tree. To look for a key k, start at the root pipe on the highest level containing a reference. Depending on the comparison with the pointed-to key k', proceed either down the pipe of k, or stay at the same level but move to the pipe of k'. To insert a new key k, find its immediate predecessor (or 'floor'), create a new pipe of the height defined by k, and (and this is the tricky part) update all the references.

[1] Note that we provide a python version of this hash-function that you can import with `from algs4.fundamentals.java_helper import java_string_hash`

Deliverables

1. `HashPipe.java` or `HashPipe.py`
2. a report (commenting on design decisions, passing tests on codeJudge, performance observations, which additional features mentioned in "Remarks" you implemented, and anything else noteworthy)

Requirements

Under the uniform hashing assumption, your update times have to be logarithmic in the number of keys.

If you use more than 100 lines of code, you are probably making a mistake.

Apart from the symbol table methods, you are required to implement a control method `public String control(String key, int h)` that returns the contents of the pipe of the given `key` at the given height h, counting from below and starting with 0. The control method returns the key that is referenced at that position, or *null*. For instance, in Figure 1, `control("H", 3)` is "P", `control("H", 2)` is "L", and `control("P", 4)` is *null*. [2] The codeJudge instance relies upon this method. Your code has to at least pass the tests in `testsSmall`. If it does not pass the large tests in `TestsBest`, most probably your insertion algorithm does not have the required logarithmic performance.

[2] This method exists partly for testing your hand-in, but it is also meant as help to you for debugging while you write your code.

Remarks

1. The intuition why the Hash Pipe is very fast is because the pipes have random heights. Of course, the hash code is not random at all, and you can easily attack the performance of this data structure using maliciously chosen keys. A better way of deciding the height of a pipe would be to use *randomness*. (For instance, the number of consecutive 'tails' in a sequence of coin flip.) However, this is a programming exercise, and we have to be able to verify your implementation, which is why we prefer a deterministic policy for determining pipe heights.

2. You have to write very little code, but it is *very hard* to get that code right. You are strongly advised to use your own `control` method to continuously test your constructions. Use pen and paper to make precise drawings for very small examples (with only 0, 1, or 2 keys in the symbol table) and use `control` to make sure that all references point to where you think they do. Figure 2 is your friend.

3. The most important design decision is how you define the inner class that represents a key–value pair (you will probably call this `Node` or maybe `Pipe`, depending on how you do it). There are at least two different ways of doing this.

4. We avoid generics in this exercise—our keys are always of type `String` and our values are always of type `Integer`. There is a reason for that[3], but if you are ambitious, you are welcome to write a generic symbol table with signature

   ```
   HashPipe<Key extends Comparable<Key>, Value>
   ```

5. If you want, you can fix the height of the root pipe (all the way to the left) to 32 (and fill the top levels with *n*ull). Otherwise, you have to rebuild the root pipe every time a new, higher, pipe enters the data structure, which involves some simple (but annoying) extra code.

6. Hint: You will probably make your life easier by having a private method like

   ```
   private Node floorNode(Key key)
   ```

   that returns the pipe of the floor of the given key. (The details depend a bit on how you chose to implement the pipes.)

7. The hash pipe really shines for range queries. If you're ambitious (and everything else really works), implement

[3] One of the ways of implementing `Node` very quickly gets you bogged down into annoying details of Java's type system for generic arrays. So this restriction makes the exercise a lot easier.

```
public Iterable<String> keys(String lo, String hi)
```

and enjoy how easy (and fast!) that was.

8. If you like binary arithmitic, you can count trailing zeros by repeated modulo and halving operations. For everybody else, there is

   ```
   Integer.numberOfTrailingZero(int i)
   ```

   in the Java standard library and `trailing_zeros` in `algs4.fundamentals.java_helper`.

9. As always, follow the coding style and naming conventions in the book.

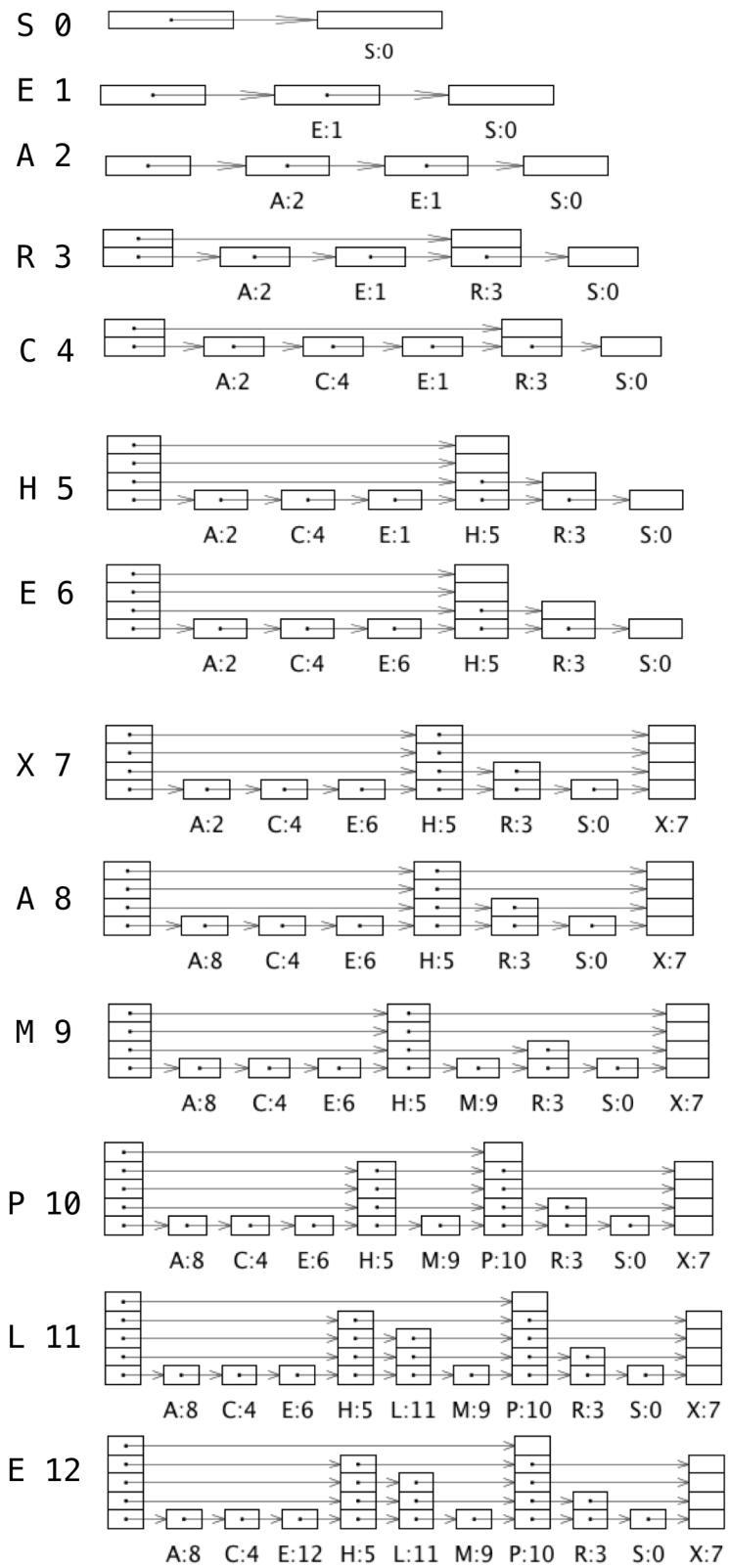

Figure 2: Trace of Hash Pipe implementation for standard symbol table indexing client.