

# HashPipe

Vlad Alexandru Ilie

April 16, 2018

Implement a symbol table using a *Hash Pipe* data structure. This mandatory individual assignment introduces another implementation of symbol tables. The main difficulty is low-level implementation choices and careful programming similar to what is needed for implementing a binary search tree of hash table.

## Deliverables

1. HashPipe.py
2. A report containing commenting on design decisions, tests passed on codeJudge, performance observations, how additional features mentioned in 'Remarks' could be implemented

## Description

The class HashPipe implements many of the methods for an ordered symbol table with String keys and Integer values. The project follows the generic ordered symbol tables implementation presented in [SW, 3.1], in addition to which two methods were created and a second class *Node* is used. The assignment has been implemented using Python but the following API represents a Java translation:

```
*public class Node // data structure for storing the key, the value and the next element
*public Node() // initializes an empty Node
*public int _value() // value getter
*public String _key() // key getter
*public Node _next() // next Node getter
*public String __repr__() // similar to public String toString()

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 Node get(String key) // the Node associated with key
public String floor(String key) // largest key less than or equal to key
*public String control(String key, Integer height) //
*public void _print() // support printing function to visualize the entire symbol table

* = differences from standard API
```

### Data structure

The *Hash Pipe* data structure is a list of pointer-based structure that represents every key–value pair using a so-called *Node*. The entry in the HashPipe at index  $i$ , represents the pointer that holds all symbols with a height greater or equal to  $i$ . The Nodes are arranged from left to right, in sorted key order. The entries of each Node point to the Node to its right, as shown in Figure 1:

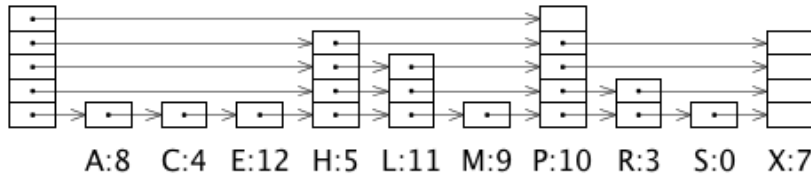


Figure 1: The data structure after running the basic symbol table client in [SW, 3.1] on input S E A R C H E X A M P L E.

```
Symbol table:
HashPipe[ 5 ]: [None:None|->|P:10|
HashPipe[ 4 ]: [None:None|->|H:5|->|P:10|->|X:7|
HashPipe[ 3 ]: [None:None|->|H:5|->|L:11|->|P:10|->|X:7|
HashPipe[ 2 ]: [None:None|->|H:5|->|L:11|->|P:10|->|R:3|->|X:7|
HashPipe[ 1 ]: [None:None|->|A:8|->|C:4|->|E:12|->|H:5|->|L:11|->|M:9|->|P:10|->|R:3|->|S:0|->|X:7|
```

Figure 2: The output after running the project on the input S E A R C H E X A M P L E.

Consider key  $k$ . The pipe’s reference at height  $h$  points to the first key  $k'$  larger than  $k$  of height at least  $h$ . There is a special *root* pipe to the left, represents the list of pointers corresponding to different heights. For example, the object at *HashPipe*[1] is an ordered linked list that holds all nodes with keys of height of at least 1. 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,<sup>1</sup> 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 an ordered linked list. A fast way of searching for a key  $k$  is to find its height  $h$  and traverse the linked list at position *HashPipe*[ $h$ ] until the key is found. Similarly, finding a key without knowing the manner in which the heights are calculated, one can simply traverse *HashPipe*[0] until the key is found. A different method of looking for a key is to start at the root pipe on the highest level containing a reference.

<sup>1</sup> Python version imported from `algs4.fundamentals.java_helper`  
`import java_string_hash`

Compare its key  $k$  with the key of the node it is pointing to  $k'$ . If  $k'$  is smaller than the desired key, proceed to the next element by staying at the same level but move to the node of key  $k'$ . If  $k'$  is greater than the desired key, move to a smaller height (proceed down the pipe of  $k$ ) and check the next Node; repeat until the looked for Node is found or all pipes are traversed.

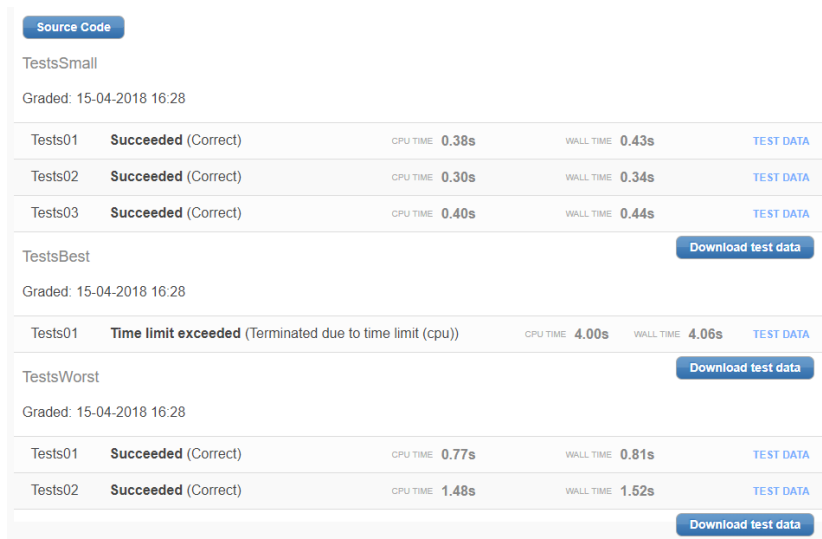
To insert a new key  $k$ , find its height so that all corresponding pipes; `HashPipe[i for i in range(0, height+1)]` can be updated. Next, for each HashPipe, find its immediate predecessor (or 'floor') and then update the references.

### Requirements

Apart from the symbol table methods, the project implements 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*. The codeJudge instance relies upon this method to check the validity of the outputs.

The project passes the *TestsSmall* and the *TestsWorst* tests on codeJudge



<a href="#">Source Code</a>				
TestsSmall				
Graded: 15-04-2018 16:28				
Tests01	Succeeded (Correct)	CPU TIME 0.38s	WALL TIME 0.43s	<a href="#">TEST DATA</a>
Tests02	Succeeded (Correct)	CPU TIME 0.30s	WALL TIME 0.34s	<a href="#">TEST DATA</a>
Tests03	Succeeded (Correct)	CPU TIME 0.40s	WALL TIME 0.44s	<a href="#">TEST DATA</a>
<a href="#">Download test data</a>				
TestsBest				
Graded: 15-04-2018 16:28				
Tests01	Time limit exceeded (Terminated due to time limit (cpu))	CPU TIME 4.00s	WALL TIME 4.06s	<a href="#">TEST DATA</a>
<a href="#">Download test data</a>				
TestsWorst				
Graded: 15-04-2018 16:28				
Tests01	Succeeded (Correct)	CPU TIME 0.77s	WALL TIME 0.81s	<a href="#">TEST DATA</a>
Tests02	Succeeded (Correct)	CPU TIME 1.48s	WALL TIME 1.52s	<a href="#">TEST DATA</a>
<a href="#">Download test data</a>				

Figure 3: codeJudge tests passed and failed

It does not pass the large tests in TestsBest, because the insertion algorithm does not have the required logarithmic performance. Unfortunately, due to the manner of nodes being implemented as Nodes with references, the complexity of inserting a node is directly proportional to the number of symbols in the table. On the other hand, once

the position on which a node has to be inserted has been found, the complexity of this operation is constant.

### Remarks

1. The `control(key, height)` method is primarily used for debugging and for checking the integrity of the data structure. The method searches in the HashPipe of corresponding height (*HashPipe[height]*) for the Node with the right key (*node = self.get(key, height)*) and unless the next Node is Null (*if(node.next == None): return None*), it returns the next Node's key (*else: return node.next.key*) *Figure 3* shows traces of the HashPipe implementation; using the input: S E A R C H E X A M P L E.
2. One of the important design decision was how to define the inner class that represents a key-value pair (in this case it is called a Node. It is a data structure defined by a key (*string*), value (*integer*), and a connection, named "*next*", to a different Node. Depending on how the attribute "*next*" is assigned, the Nodes form ordered linked lists, which together form the individual entries / pipes in the *HashPipe* data structure.
3. In order to avoid rebuilding the root pipe every time new, higher nodes are added; the root pipe is initialized as a list of 32 empty Nodes.
4. A method similar to *private Node floorNode(Key key)* was implemented in Python as *floor(self, key, height)*. The method returns the first Node, at the specified height, with a key smaller than the parameter key.
5. In order to find the height of a key, the project imports the methods `java_string_hash` and `trailing_zeros` from `algs4.fundamentals.java_helper`. The underlying logic of this operation is to transform the string into ASCII code, subsequently transform it into a binary representation and lastly, count the number of trailing zeros. In practice, this is achieved with the following line of code `height = trailing_zeros(java_string_hash(key))`

```

Inserting : ( S , 0 )
  HashPipe[ 1 ] : |None:None|->|S:0|
Inserting : ( E , 1 )
  HashPipe[ 1 ] : |None:None|->|E:1|->|S:0|
Inserting : ( A , 2 )
  HashPipe[ 1 ] : |None:None|->|A:2|->|E:1|->|S:0|
Inserting : ( R , 3 )
  HashPipe[ 2 ] : |None:None|->|R:3|
  HashPipe[ 1 ] : |None:None|->|A:2|->|E:1|->|R:3|->|S:0|
Inserting : ( C , 4 )
  HashPipe[ 2 ] : |None:None|->|R:3|
  HashPipe[ 1 ] : |None:None|->|A:2|->|C:4|->|E:1|->|R:3|->|S:0|
Inserting : ( H , 5 )
  HashPipe[ 4 ] : |None:None|->|H:5|
  HashPipe[ 3 ] : |None:None|->|H:5|
  HashPipe[ 2 ] : |None:None|->|H:5|->|R:3|
  HashPipe[ 1 ] : |None:None|->|A:2|->|C:4|->|E:1|->|H:5|->|R:3|->|S:0|
Inserting : ( E , 6 )
  HashPipe[ 4 ] : |None:None|->|H:5|
  HashPipe[ 3 ] : |None:None|->|H:5|
  HashPipe[ 2 ] : |None:None|->|H:5|->|R:3|
  HashPipe[ 1 ] : |None:None|->|A:2|->|C:4|->|E:6|->|H:5|->|R:3|->|S:0|
Inserting : ( X , 7 )
  HashPipe[ 4 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:2|->|C:4|->|E:6|->|H:5|->|R:3|->|S:0|->|X:7|
Inserting : ( A , 8 )
  HashPipe[ 4 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:8|->|C:4|->|E:6|->|H:5|->|R:3|->|S:0|->|X:7|
Inserting : ( M , 9 )
  HashPipe[ 4 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:8|->|C:4|->|E:6|->|H:5|->|M:9|->|R:3|->|S:0|->|X:7|
Inserting : ( P , 10 )
  HashPipe[ 5 ] : |None:None|->|P:10|
  HashPipe[ 4 ] : |None:None|->|H:5|->|P:10|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|P:10|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|P:10|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:8|->|C:4|->|E:6|->|H:5|->|M:9|->|P:10|->|R:3|->|S:0|->|X:7|
Inserting : ( L , 11 )
  HashPipe[ 5 ] : |None:None|->|P:10|
  HashPipe[ 4 ] : |None:None|->|H:5|->|P:10|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|L:11|->|P:10|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|L:11|->|P:10|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:8|->|C:4|->|E:6|->|H:5|->|L:11|->|M:9|->|P:10|->|R:3|->|S:0|->|X:7|
Inserting : ( E , 12 )
  HashPipe[ 5 ] : |None:None|->|P:10|
  HashPipe[ 4 ] : |None:None|->|H:5|->|P:10|->|X:7|
  HashPipe[ 3 ] : |None:None|->|H:5|->|L:11|->|P:10|->|X:7|
  HashPipe[ 2 ] : |None:None|->|H:5|->|L:11|->|P:10|->|R:3|->|X:7|
  HashPipe[ 1 ] : |None:None|->|A:8|->|C:4|->|E:12|->|H:5|->|L:11|->|M:9|->|P:10|->|R:3|->|S:0|->|X:7|

```

Figure 4: Visualization of the symbol table for each addition, using the input SEARCHEXAMPLE