

# DATA-Lab4

## Question 2.

To assess the collision rate in the dictionary, we need to define some parameters in the dictionary:

- Number of items to be stored:  $n=10^3$
- Range of keys:  $0 \leq k \leq 10^4$
- Number of available slots in the dictionary:  $m=20$

$$H_0(k) = k \% 20$$

On avg, the number of items in each slot =  $10^3/20 = 50$

Collision rate =  $50/10^3=0.05$

If we still use hash function in form of  $H(k)=k\%m$ , the collision rate can simply be reduced by increasing  $m$  until  $m \geq \max \text{key in the table}$ . Then, each slot stores only 1 item. However, if we do so, we are back to "Direct Addressing", which is impractical for great number of elements and key with great values.

Even we use UHS, we still need to map  $(ax+b)\%p$  to  $Z_m$ . Then the collision rate is still at max  $1/m=0.05$ .

To check accuracy of this, I choose a random  $H_1(k)$  to compare the collision rate with  $H_0(k)$

$$H_1(k) = ((4x + 1)\%10007)\%20$$

2-tailed Wilcoxon Rank Sum is used with null hypothesis is there is no significance difference in the collision distribution when using ***hash\_function\_division()*** and ***hash\_function\_universal()***

```
Attempt no.1: p_value=0.81; H0 NOT rejected
Attempt no.2: p_value=0.71; H0 NOT rejected
Attempt no.3: p_value=0.81; H0 NOT rejected
Attempt no.4: p_value=0.87; H0 NOT rejected
Attempt no.5: p_value=0.98; H0 NOT rejected
```

## Question 5.

$$n=(2^h)-1 \Leftrightarrow h = \log_2(n + 1)$$

The algorithm works in  $O(\text{tree\_height})=O(\log_n)$ . In each iteration, we go down a level in the tree and only visit a node of that level. Then, in the worst case, when there is no node with matching key in the bst and the checking procedure is on the longest branch of the tree, we need to go from top to the bottom of the tree. In other words, the time taken depends on the height of the tree.

However, in the worst case, where all number is inserted to a singly branch of the tree as below, the height of the tree becomes  $n$ . Then `iterative_tree_search()` works in  $O(n)$

```
(10)---NIL
      |
      |_(11)---NIL
            |
            |_(12)---NIL
                  |
                  |_(13)---NIL
                        |
                        |_...
```

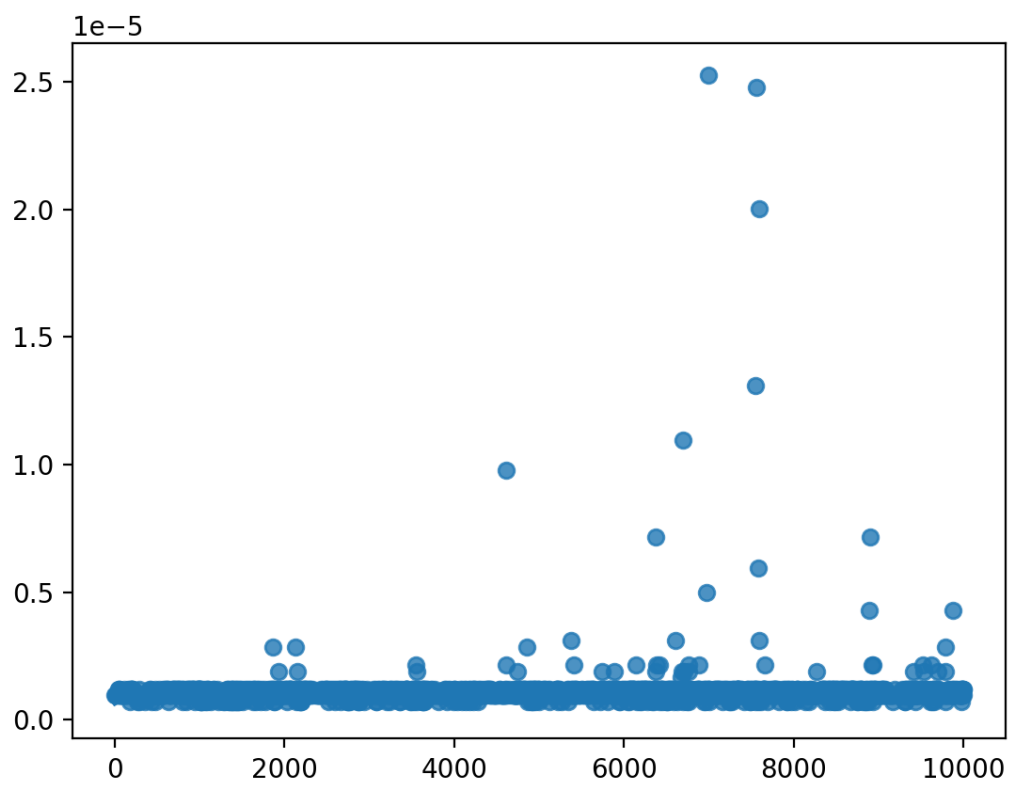


Figure3.1: Running time of `bst_iterative_search()` at different size of input

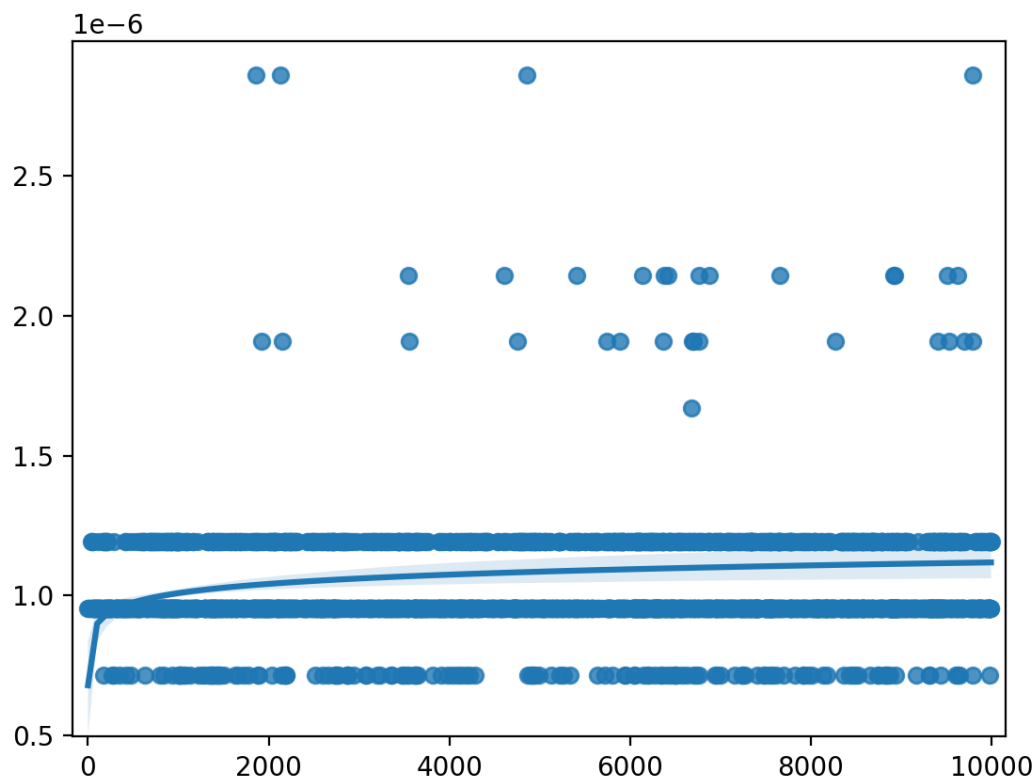


Figure3.2: Running time of `bst_iterative_search()` at different size of input (Zoomed)

singly\_linked\_list:  $O(n)$

doubly\_linked\_list:  $O(n)$

array:  $O(n)$

To compare the efficiency in run time of `search()` in bst, array, singly linked list (sll) and doubly linked list (dll), 2 tests were performed:

1. Running time of searching for a number not in bst, array sll and dll at an input size of 200
2. **One-sided Wilcoxin for different input size ( $n \leq 2000$ ):**  $H_0$ : Searching for an element in a binary search tree takes less time than in array/singly linked list/doubly linked list  
(The median difference in runtime of 2 implementations ( $T_{bst} - T_{another\_function}$ ) is negative)

⇒ Both test show that searching for an element in bst takes less time than in each of array, singly linked list and doubly linked list.

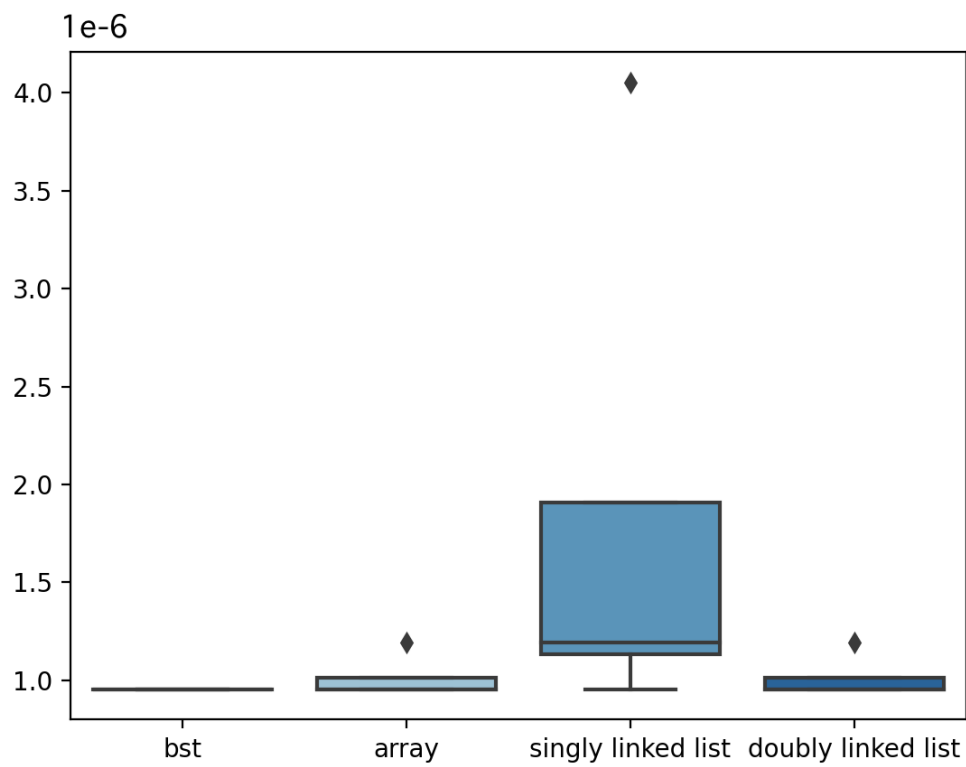


Figure 2. Running time of search() function in bst, array, singly linked list and doubly linked list, n=200

```

Attempt no.1
  p_value_bst_arr: 0.1464
  p_value_bst_sll: 0.9722
  p_value_bst_dll: 0.7094
Attempt no.2
  p_value_bst_arr: 0.1028
  p_value_bst_sll: 0.9682
  p_value_bst_dll: 0.9953
Attempt no.3
  p_value_bst_arr: 0.0846
  p_value_bst_sll: 0.8131
  p_value_bst_dll: 0.6849
Attempt no.4
  p_value_bst_arr: 0.0514
  p_value_bst_sll: 0.9956
  p_value_bst_dll: 0.1790

```