**Report for HW2** -Tashi Tengyal (2020-15441)

**Environment**

Python 3

**Directions to run the program**

First, make sure you are in the CORRECT directory. In your CLI, run <python main.py> and follow the instructions.

Upon correct input entry, output and checker files for the respective input file is created in the directory. Checker file shows the correctness of the program and it also displays the time taken to finish running all the operations in the input sequence.

Input1 has input.txt with 50 inputs.

Input2 has input.txt with 100,000 inputs.

**Main Program**

After the reading the input file, the program generates an input list of lists (input\_arr) that contains [Operation, (Key or Rank)]. To execute the specified operation, the main program takes in the 1st element of the list above and decides the type of operation to be performed. The return value of each operation is stored in an output\_arr.

**Checker Program**

The checker program first takes in the input\_arr obtained from the input file. checker\_arr is used to contain elements that are the return values from the 4 functions: check\_insert(), check\_del(), check\_rank() and check\_del().

1. check\_insert(insert\_arr, key):

if(key not in insert\_arr):

insert\_arr.append(key)

return insert\_arr

else: return [0]

🡪Time Complexity = O(n)

1. check\_del(insert\_arr, del\_arr, key):

if(key in insert\_arr):

remove key from insert\_arr

del\_arr.append(key)

return del\_arr

else: return [0]

🡪Time Complexity = O(n)

1. check\_sel(insert\_arr, sel\_arr, rank):

if(rank <= len(insert\_arr)):

tmp = insert\_arr.sort()

sel\_arr.append(tmp[rank-1])

return sel\_arr

else: return [0]

🡪Time Complexity = O(nlogn) #Python sort() uses TimSort Algorithm.

1. check\_rank(insert\_arr, rank\_arr, key):

if(key in insert\_arr):

count = 0

for(i=0; i<=len(insert\_arr); i++):

if(insert\_arr[i]<= x):

count++

rank\_arr.append(count)

return rank\_arr

Time Complexity = O(n)

To make sure that the checker\_arr follows the same format as the output\_arr, only the last element from return value(i.e an array corresponding to the operations above) is appended to it. For example,

After check\_insert(insert\_arr, 1) 🡪 insert\_arr = [1], checker\_arr = [1]

After check\_insert(insert\_arr, 2) 🡪 insert\_arr = [1, 2], checker\_arr = [1, 2]

After check\_delete(insert\_arr, del\_arr, 2)🡪insert\_arr = [1], del\_arr = [2],

checker\_arr = [1, 2, 2]

After inserting 3, 4, 5, 6 🡪 insert\_arr = [1, 3, 4, 5, 6] & checker\_arr = [1, 2, 2, 3, 4, 5, 6]

After check\_sel(insert\_arr, sel\_arr, 5) 🡪 insert\_arr = [1, 3, 4, 5, 6], sel\_arr = [6],

checker\_arr = [1, 2, 2, 3, 4, 5, 6, 6]

After check\_rank(insert\_arr, rank\_arr, 5) 🡪insert\_arr=[1, 3, 4, 5, 6], rank\_arr = [5],

checker\_arr = [1, 2, 2, 3, 4, 5, 6, 6, 5]

Finally, to check the correctness of the entire algorithm, the checker\_arr and output\_arr are compared. If they are equal then the checker.txt displays “CORRECT”

if(output\_arr == checker\_arr):

return True

else: return False

**Time Complexity Measurements**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size | os\_insert (s) | os\_delete (s) | os\_select (s) | os\_rank (s) |
| 1000 | 0.0619549751282 | 0.0602900981903 | 0.0504939556122 | 0.0572068691254 |
| 0.00650095939636 | 0.00654411315918 | 0.00666689872742 | 0.00664496421814 |
| 5000 | 0.246812105179 | 0.244976043701 | 0.24440908432 | 0.242524147034 |
| 0.114891052246 | 0.115503072739 | 0.110481977463 | 0.114362001419 |
| 10000 | 0.442445993423 | 0.448291778564 | 0.444292783737 | 0.467293024063 |
| 0.340669155121 | 0.341927051544 | 0.349743843079 | 0.335760116577 |
| 15000 | 0.635651111603 | 0.665100097656 | 0.634196043015 | 0.635662078857 |
| 0.627089977264 | 0.641180038452 | 0.628936052322 | 0.634447097778 |
| 100000 | 2.84124898911 | 2.83054304123 | 2.78254008293 | 2.86695218086 |
| 5.66241002083 | 5.58612394333 | 5.7319829464 | 5.7376730442 |

Blue- Time for Main Program Orange- Time for Checker Program

**As per the measurements above, the Main program takes about O(logn) and the Checker Program takes about O(n).**

As evident from the table above, RB tree seems to become very efficient compared to the Checker program as the input size increases.