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().

```
i.      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)

```
ii.     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)

```
iii.    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.

```
iv.     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.