2022_DS_Fall_Exercises 4

Exercise #22 *

[Programming Project]

The objective of this assignment is to come up with a composite sorting function that is good on the worst-time criterion. The candidate sort methods are

1. Insertion sort,
2. Qucik sort,
3. Merge sort,
4. Heap sort

To begin with, program these sort methods in C. In each case, assume that n integers are to be sorted.

- In the case of qucik sort, use the median-of-three method.
- In the case of merge sort, use the iterative method(as a separate exercise you might wish to compare the runtimes of the iterative and recursive versions of merge sort and determine what the recursion penalty is in your favorite language using your favorite compiler).

Check out the correctness of the programs using some test data. Since quite detailed and working functions are given in the book, this part of the assignment should take little effort. In any case, no points are earned untile after this step.

To obtain reasonably accurate runtimes, you need to know the accuracy of the clock or timer you are using. Determine this by reading the appropriate manual. Let the clock accuracy be δ. Now, run a pilot test to determine ballpark times for your four sorting functions for $n = 500, 1000, 2000, 3000, 4000,$ and may not be much larger than the clock accuracy.

To *time an event that is smaller than or near the clock accuracy, repeat it many times and divide the overall time by the number of repetitions*. You should obtain times that are accurate to within 1%.

We need worst-case data for each of the four sort methods. The worst-case data for insertion sort are easy to generate. Just use the sequence $n, n-1, n-2, \ldots, 1$. Worst-case data for merge sort can be obtained by working backward. Begin with the last merge your function will perform and make this work hardest. Then look at the second-to-last merge, and so on. Use thise logic to obtain a program that will generate worst-case data for merge sort for each of the above values of n.

Generating worst-case data for heap sort is the hardest, so here we shall use a random permutation generator (one is provide in Program 7.18). We shall generate random permutations of the desired size, clcok heap sort on each of these, and use the max of these times to approximate to the worst-case time. You will be able to use more random permutations for smaller values of n than for larger. For no value of n should you use fewer than 10 permutations. Use the same technique to obtain worst-case times for quick sort.

Having settled on the test data, we are ready to perform our experiment.

```
void permute(element a[], int n) {
    /* random permutation generator */
    int i, j;
    element temp;
    for(i = n; i >= 2; i--) {
        j = rand() % i + 1;
        SWAP(a[j], a[i], temp);
    }
}
```

Program 7.18: Random permutation generator

Obtain the worst-case times. From these times you will get rough idea when one function performs better than the other. Now, narrow the scope of your experiments and determine the exact value of n when one sort method outperforms another. For some methods, this value may be 0. For instance, each of the other three methods may be faster than quick sort for all values of n.

Plot your findings on a single sheet of graph paper. Do you see the n^2 behavior of insertion sort and qucik sort and the $n \log n$ behavior of the other two methods for suitably large n (about $n > 20$)? If not, there is something wrong with your test or your clcok or with both. For each value of n determine the sort function that is fastest (simply look at your graph). Write a composite function with the best possible performance for all n. Clock this function and plot the times on the same graph sheet you used earlier.

WHAT TO TURN IN

You are required to submit a report that states the

1. clock accuracy
2. the number of random permutations tried for heap sort
3. The worst-case data for merge sort and how you generated it
4. A table of times for the above values of n, the times for the narrowed ranges, the graph, and a table of times for the composite function.
5. In addition, your report must be accompanied by a complete listing of the program used by you (this includes the sorting functions and the main program for timing and test-data generation).

Exercise #23 *

Write a function to delete the pair with key k from a hash table that uses linear probing. Show that simply setting the slot previously occupied by the deleted pair to empty does not solve the problem. How must Get(Program 8.3) be modified so that a correct search is madee in the situation when deletions are permitted? Where can a new key be inserted?

Technical Specification

You should write the functions, which are including insert, delete, and search. You must guarantee the 3 functions are working.

Exercise #24

Write a function to delete the element with key k from a Patricia. The complexity of your function should be $O(h)$. where h is the height of the Patricia instance.

Input Format

First line consists of one number b, it represents the length of every key in the following input.

After first line, there will be at most 1000 lines of input text. Each line describes one Patricia operation.

- $\text{insert} <\text{key}> <\text{value}>$
  This Patricia operation declares one key-value insertion.
  - key will be a string with length b, and its content will only be '0' or '1'.
  - value will be an integer.
- $\text{search} <\text{key}>$
  This Patricia operation declares one key search operation.
- $\text{delete} <\text{key}>$
  This Patricia operation declares one key-value deletion.
- quit
  No more input, exit the program.

Output Format

- $\text{insert} <\text{key}> <\text{value}>$
  - output `insert -> [value]` if the insert operation is successful.
  - output `insert -> conflict` if the given key already exists in Patricia.
- $\text{search} <\text{key}>$
  - output `search -> [value]` if the corresponding key-value pair is found.
  - output `search -> not found` if the given key doesn't exist in Patricia.
- $\text{delete} <\text{key}>$
  - output `delete -> [value]` if the corresponding key-value pair is found and deleted.
  - output `delete -> not found` if the given key doesn't exist in Patricia.

Technical Specification

- The input will consist of at most 1000 Patricia operations.
- $1 \leq b \leq 10^3$
- For every key declares in the insert/search/delete operation, $|\text{key}| = b$. The content of key will only be '0' or '1'.
- For every value declares in the insert operation, $0 \leq \text{value} \leq 10^6$
- The complexity of your Patricia delete function should be $O(h)$.

Sample Input

```
7
insert 0010100 1
search 0010100
delete 0010100
quit
```

```
5
insert 10101 1
insert 10101 1
search 10101
delete 10101
search 10101
delete 10101
quit
```

Sample Output

```
insert -> 1
search -> 1
delete -> 1
```

```
insert -> 1
insert -> conflict
search -> 1
delete -> 1
search -> not found
delete -> not found
```