EE441 DATA STRUCTURES
PROGRAMMING ASSIGNMENT #3

Due Date: January 3, 2021 , 23:55

For your questions: Kamil SERT - ksert@metu.edu.tr

Part 1 – HASHING

In this part of the homework you will implement a hash table data structure that is utilized to insert, delete, and search for keys. The keys will be T.C. Identification Number (an eleven digit number). The hash table is going to be an array whose elements are pointers to a linked list (or simply NULL if the corresponding hash table entry is empty). If collisions occur, you should use chaining with linear probing for collision resolution (hash table entries must hold a key part and a link part). You will need to implement the following 3 hash functions:

- **Folding:** The key is simply divided into sections that are 3 digits long, and these sections are added together. The final hash value is (sum mod N) where N is size of the hash table. For example, if key=15726349053 and N=100, we divide k into 4 sections: 157, 263, 490, 53. Sum of these sections is 963 and the hash value is 63 for this key.
- **Middle Squaring:** Take middle 3 digits of the key and square them. The final hash value is (sum mod N) where N is size of the hash table. For example, if key=15726349053 and N=100, the middle 3 digits is 634. Square of 634 is 401,956 and the hash value is 56 for this key.
- **Truncation:** Delete the first 9 digits and use only the last 2 digits of the key. The final hash value is (sum mod N) where N is size of the hash table. For example, if key=15726349053 and N=100, the last two digits is 53 and the hash value is 53 for this key.

You should also implement a menu driven user interface to perform operations on hash table. The menu should look like the following;

1. Initialize Hash Table
2. Load T.C. ID Numbers from file
3. Add new T.C. ID Number
4. Delete a T.C. ID Number
5. Search for a T.C. ID Number
6. Print out Hash Table

Operation 1: Asks size of the hash table 'N' and the hash function (folding, middle squaring, or truncation) to user and initializes hash table of size N. All elements will be empty initially.

Operation 2: Loads T.C. ID Numbers from file named as 'hash_table_init'. This function will be used for test purposes. Also this operation makes your work easier.
Operation 3: Adds new T.C. ID Number to hash table. Reinserting a number is not possible.
Operation 4: Deletes a T.C. ID Number from hash table.
Operation 5: Searches for a T.C. ID Number in the hash table.
Operation 6: Prints out hash table in easy-to-read format. For this operation the output can look like the following;

1.74521647384
2.65634823940
3. E
4. D
5.98246324747
6. E
7. E
8. E
9.27583294005
10. E

Make sure you handle possible error conditions for these operations.

Performance Evaluation
For this part of the homework you are going to measure the loading factor and the number of collisions for folding, middle squaring, and truncation. The details are given below;

Repeat the following steps for each of the hash functions.
1. Initialize hash table of size 100.
2. Generate 200 T.C. IDs randomly.
3. Add these numbers one by one to the hash table.

Answer the following questions;
1. What is the loading factor for folding, middle squaring, and truncation hash functions?
2. What is the number of collisions for folding, middle squaring, and truncation hash functions?
3. Compare these hash functions.

Part 2 – SORTING

In this part of the homework you will analyze the performance for some of the sorting algorithms. Sorting is one of the most fundamental issues within computer science. In this assignment you are going to implement several sorting algorithms and gather statistics (including number of data comparison, number of data moves, and the total execution time for the sort) about these algorithms.

You are going to implement the following sorting algorithms;

- Bubble Sort
- Selection Sort
- Quick Sort-1 (select first element as pivot)
- Quick Sort-2 (select middle element as pivot)
- Quick Sort-3 (select randomly chosen element of the array – array (random_index) – as pivot)
- Quick Sort-4 (select the median of 3 randomly chosen elements of the array as pivot)
- Your Own Algorithm

How to Develop Your Own Algorithm:
When the array size gets large, quick sort is clearly the fastest sorting algorithm. However, when the array size is small enough, quick sort runs slower ($\Theta(n^2)$). When sorting a large array with quick sort, we need to sort many small subarrays. While sorting one small array with a faster algorithm is negligible, sorting hundreds of small arrays with a faster algorithm can make a difference in the overall performance of the sort. For this part of the assignment, we ask you to develop the fastest possible sorting algorithm by combining quick sort with another sorting algorithm.

You have several options;
- Use quick sort until the array gets small enough, and then use another sorting algorithm to sort the small lists.
- Use quick sort to mostly sort the array, i.e. use quick sort to sort the array until a defined cutoff size is reached (the array is going to be mostly sorted). You can use another sorting algorithm on the entire array to quickly complete the sorting.
- Design your own method.

You must explain your own algorithm in detail.

You should also implement a menu driven user interface to perform some operations. The menu should look like the following;
1. Initialize input array randomly
2. Load input array from a file
3. Perform Bubble Sort
4. Perform Quick Sort
5. Perform Selection Sort
6. Perform Your Own Sort
7. Compare sorting algorithms

Operation 1: Asks size of the array 'N' to user and initializes array which is going to be sorted of size N with random numbers.
Operation 2: Loads array from file named as 'input_array'. This function will be used for test purposes. Also this operation makes your work easier.
Operation 3: Performs Bubble Sort on the array and prints sorted array in easy-to-read format.
Operation 4: Performs Quick Sort on the array and prints sorted array in easy-to-read format. This operation also asks the user how the pivot is selected.

Operation 5: Performs Selection Sort on the array and prints sorted array in easy-to-read format.
Operation 6: Performs your own sorting algorithm on the array and prints sorted array in easy-to-read format.
Operation 7: Runs all the 7 algorithms and prints out the statistics about these algorithms for the same input array. These statistics includes the number of data comparisons, the number of data moves, and the time required for the sort.

You should be careful about measuring the elapsed time. You should not count the time for reading the array, but you should count the time to sort the array. In order to measure elapsed time of program you can use the standard clock function, which is exported by the standard **ctime** interface. The clock function returns the amount of time the processing unit of the computer has used in the execution of the program. You can always convert the system-dependent clock units into seconds by using the following expression:

```
double(clock()) ; clocks per second
```

If you record the starting and finishing times in the variables start and finish, you can use the following code to compute the time required by a calculation:

```
#include <ctime>
int main()
{
        …
        double start = double(clock());
        // perform sorting the array
        double finish = double(clock());
        double elapsed_time = finish - start;
        …
}
```

There is no guarantee that the system clock unit is precise enough to measure the elapsed time. For example, if you use this strategy to measure elapsed time to sort 10 integers, the elapsed time value at the end of the code fragment would be 0. The reason is that the processing unit on most machines can execute many instructions in a single clock tick. One way to overcome this problem is to repeat the calculation many times between the two calls to the clock function. For example, if you want to determine how long it takes to sort 10 numbers, you can perform the sort 10 numbers 100 times in a row and then divide the total elapsed time by 100. This method gives you a timing measurement that is much more accurate. You will figure out how many times you need to perform the sort operation for different-sized inputs to obtain accurate results.

The output for this operation can look like the following;

```
Array size: 1000 numbers

Algorithm     #comparisons        #moves        time (msec)

Bubble        ..................  ..........    ..................
Quick         ..................  ..........    ..................
Selection     ..................  ..........    ..................
```

Make sure you handle possible error conditions for these operations.

Performance Evaluation

For this part of the homework we want from you to write a one or two pages report. You need to run your program for different sized arrays and collect results. The details are given below;

- Create 5 arrays of size 100 whose elements are randomly initialized. Sort these arrays and calculate average number of comparisons, number of data moves and elapsed time.

```
for(int i=0;i<5;i++)
{
        int size=100;
        int randArray[size];
        for(int j=0; j<size; j++)
        randArray[j]=rand()%(10*size); //Generate numbers between 0 to (10*size)
        // sort the array (insert your sorting function here)
        // collect statistics
        i++;
}
```

- Create 5 arrays of size 1,000 whose elements are randomly initialized. Sort these arrays and calculate average number of comparisons, number of data moves and elapsed time.
- Create 5 arrays of size 5,000 whose elements are randomly initialized. Sort these arrays and calculate average number of comparisons, number of data moves and elapsed time.
- Create 5 arrays of size 10,000 whose elements are randomly initialized. Sort these arrays and calculate average number of comparisons, number of data moves and elapsed time.
- Create 5 arrays of size 25,000 whose elements are randomly initialized. Sort these arrays and calculate average number of comparisons, number of data moves and elapsed time.

Provide these results in a table and in a graph, and compare these algorithms in detail in your report.

Submission

- You should insert comments to your source code at appropriate places without including any unnecessary detail. **Comments WILL be graded.**
- Submit the whole Code::Blocks project folders and your report (.pdf format) in a zip file. The file name should be "**EE441_PA3_firstname_lastname_studentID.zip**".
- Indicate how much time you spent for this assignment in your report.
- Late submissions are welcome, but penalized according to the following policy:
    - 1 day late submission: HW will be evaluated out of 70 points
    - 2 days late submission: HW will be evaluated out of 50 points
    - 3 days late submission: HW will be evaluated out of 30 points
    - 4 or more days late submission: HW will not be evaluated