Data Structure And Algorithms - Coursework

Vishnu Sreekumar

January 2018

Contents

1 Analysis of the problem

T he goal of the exercise is to store 10 years worth of 5 minute interval sensor readings in such a way that lookups can be done efficiently. One main characteristic of the data that is to be processed is that the storage requirements are constant. ie it is given that the data is for 10 years and every year has 12 months. The number of days in every month is constant (though it changes across months) and so is the total number of readings per sensor per day (One reading per 5 minutes - 288 readings per sensor per day).

A common requirement in the problem set is the ability to get the value, given a sensor type and date (year, month and/or day). One of the questions also asks for the exact time stamp of the reading.

Values like maximum, average and total can be calculated along with inserts to avoid redundant linear parsing of sensor readings, hence the proposed data structure should support this. The fact that the given data set may contain duplicates has to be considered while searching or sorting the readings and have to choose search and sort algorithms that can handle duplicates.

Each input file has an year worth of data in chronological order. It is assumed that that files are going to be processed line by line, while inserting various readings to it's proposed data structure.

2 Proposed Data Structures

Since the keys of the input data (Year, Month, Day, Sensor Type) are pre-known and definite, one of the best options to store them is a *Map Abstract Data Type*. A map ADT lets us do the lookups with constant time complexity, ie O(1). [1]

For example to get the object of class data structure for Wind Speeds on 2nd February 2014 a single lookup over the map ADT fetches the required daily reading data structure.

READINGS\[2014\]\[FEB\]\[2\]\[S\] -> Object of DailyReadings

The below diagram shows a visualization of the above call with all the data structures included.

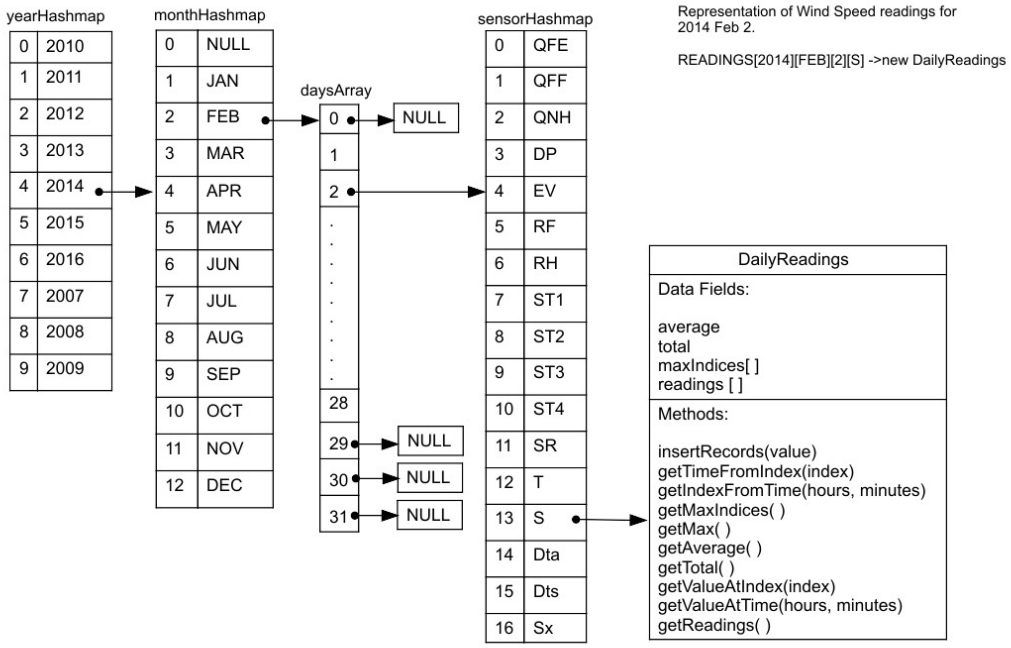

Figure 1: Wind Speeds on 2nd February 2014

Individual data structures in the above diagram are discussed in detail below.

2.1 Year Map ADT

Data for a year is stored in the *yearHashmap*. Since the data set is smaller (10 entries), a simple modular hashing function with a linear collision resolution can be used on an array of 10 elements.

$h(k) = k \mod m$

where k = year and m = 10.

2.1.1 Lookup table

Key	Index
2007	7
2008	8
2009	9
2010	0
2011	1
2012	2
2013	3
2014	4
2015	5
2016	6

Element of yearHashmap is an object of monthHashmap.

yearHashmap[2014] –> monthHashmap[]

2.2 Month Map ADT

The *monthHashmap* uses a custom hashing function which accepts the 3 letter string representation of month name and output the numeric value of the month. The output of this hashing function is always unique, hence there is no need for a collision resolution function. The size of the array used by this hashmap is 13 and index 0 is pointed to NULL. So that the actual values begins from 1 and goes up-to 12.

h(JAN) = 1
h(FEB) = 2
h(MAR) = 3
h(APR) = 4
h(MAY) = 5
h(JUN) = 6
h(JUL) = 7
h(AUG) = 8
h(SEP) = 9
h(OCT) = 10
h(NOV) = 11
h(DEC) = 12

2.2.1 Lookup Table

Key	Index
JAN	1
FEB	2
MAR	3
APR	4
MAY	5
JUN	6
JUL	7
AUG	8
SEP	9
OCT	10
NOV	11
DEC	12

Element of monthHashmap is a daysArray.

monthHashmap[FEB] -> daysArray[]

2.3 Days Array

The *daysArray* is an array of *sensorHashmap* values. Use of a map ADT here will be redundant and the day number can be used as the index of the array without passing it through a hashing function. The size of the array is 32, indexed (0..31), with all values initialized to NULL.

daysArray = [NULL, NULL, \ldots, NULL]

Initializing the array elements to NULL helps in handling the varying number of days across months. For example if the daysArray[31] is pointing to NULL, it can be safely assumed that the month has only 30 days. However while looping through the daysArray, we need to keep a separate counter variable to get the actual number of days at the end.

Algorithm 2.1: Count number of days in a given daysArray.

```
1 SET counter = 0
2 FOR index IN 1 TO 31:
3 IF daysArray[index] NOT NULL:
4 SET counter = counter + 1
5 ENDIF
6 ENDFOR
```

Element of daysArray is a sensorHashmap.

2.4 Sensor Map ADT

The *sensorHashmap* uses a custom hashing function as well. It takes the sensor type as an input and returns the index.

$h(\mathrm{QFE}) = 0$
$h(\mathrm{QFF}) = 1$
$h(\mathrm{QNH}) = 2$
$h(\mathrm{DP}) = 3$
$h(\mathrm{EV}) = 4$
$h(\mathrm{RF}) = 5$
$h(\mathrm{RH}) = 6$
$h(\mathrm{ST1}) = 7$
$h(\mathrm{ST2}) = 8$
$h(\mathrm{ST3}) = 9$
$h(\mathrm{ST4}) = 10$
$h(\mathrm{SR}) = 11$
$h(\mathrm{T}) = 12$
$h(\mathrm{S}) = 13$
$h(\mathrm{Dta}) = 14$
$h(\mathrm{Dts}) = 15$
$h(\mathrm{Sx}) = 16$

2.4.1 Lookup Table

Key	Value
QFE	0
QFF	1
QNH	2
DP	3
EV	4
RF	5
RH	6
ST1	7
ST2	8
ST3	9
ST4	10
SR	11
T	12
S	13
Dta	14
Dts	15
Sx	16

Element of a sensorHashmap is an object of DailyReadings ADT.

2.5 Daily Readings

The goal in the design of data set for daily readings is to do the following actions efficiently:

1. Insert new item
2. Find the max value(s) including duplicates along with the timestamp
3. Find average value
4. Find total

We need to use a custom Abstract Data Type [1] with sensor readings, max, average and total values as data members and methods to achieve the above requirements.

Daily sensor readings are inserted to an array data field in this abstract data structure. Max value(s), average and total can be calculated or updated along with insert operation. Average and total are also stored as data fields. However we need some special considerations for storing the Max values.

It is a requirement that we not only need the maximum values but also the time associated with the values. We know that readings are taken every 5 minutes and that makes total 288 readings per day. If we get the values sorted by time and store these values in an array of size 289 (ie index 0 - 288) starting from index 1, the nth item will be the reading at (n * 5)th minute.

In 24 hr format we usually represent time as 00:00 to 23:59. That is the 288th reading in our case should actually be the first reading of the next day. The below algorithm can be tweaked to make it more realistic and support this edge case. However, for the sake of simplicity, we are assuming that the first reading of the day is taken at 00:05 and the last reading at 24:00. Given an index n, we can find the time as below.

$totalMinutes = n * 5$
$hours = |totalMinutes/60|$
$minutes = |totalMinutes \ mod \ 60|$

For example:
readings[15] is the reading at 75th minute or in other words, the time will be 01:15 (24 hr format). The calculation can be reversed to find the reading for a specific time as follows.

$totalMinutes = (hours * 60) + minutes$
$n = totalMinutes/5$
$value = readings[n]$

Since we can easily find the value and time from the index of the item in the sensor readings array, the Max values can be represented as an array of their indices in the sensor readings array.

An abstract data structure of the following blue print can be used to store the daily readings.

2.5.1 Data fields

- Array of values
- Array of indices of max values
- Average
- Total

2.5.2 Methods

- Insert new reading
- Return array of readings
- Return max values array
- Return max value
- Return total
- Return average
- Return time given an index
- Return index given a time

2.5.3 Pseudo Code

Algorithm 2.2: Blueprint of dailyReadings ADT

```
1         CLASS DailyReadings():
2         SET maxIndices = []
3         SET average = 0.0
4         SET total = 0.0
5         SET readings = []
6
7         FUNCTION insertRecord(value):
8         PUSH value TO readings[]
9         SET lastIndex = (length of readings[]) - 1
10
11        IF (length of maxIndices[]) > 0:
12        // Get the value at current max index
13        SET currentMax = readings[maxIndices[0]]
14
15        IF readings[lastIndex] > currentMax:
16        // Re-initialize the maxIndices array with lastIndex as the only member
17        SET maxIndices = [lastIndex]
18        ELSEIF readings[lastIndex] == currentMax:
19        // Push lastIndex to maxIndices array if value is equal to the max value (duplicate)
20        PUSH lastIndex TO maxIndices[]
```

```
21          ENDIF
22
23          ELSE:
24          // If max indices array is empty add this one.
25          PUSH lastIndex TO maxIndices
26          ENDIF
27
28          SET total = total + value
29          SET average = total / (length of readings[])
30          ENDFUNCTION
31
32          FUNCTION getTimeFromIndex(index):
33          //Assuming that time is the string representation of HH:MM in 24 hours format
34
35          SET totalMinutes = index * 5
36          SET hour = INTEGER(totalMinutes / 60)
37          SET minutes = INTEGER(totalMinutes % 60)
38
39          RETURN hour + ":" + minutes
40          ENDFUNCTION
41
42          FUNCTION getIndexFromTime(hours, minutes):
43          SET totalMinutes = (hours * 60) + minutes
44          SET index = totalMinutes / 5
45          RETURN index
46          ENDFUNCTION
47
48          FUNCTION getMaxIndices():
49          RETURN maxIndices
50          ENDFUNCTION
51
52          FUNCTION getMax():
53          //Return the numeric max value (no duplicates)
54          IF (length of maxIndices) > 0:
55          RETURN readings[maxIndices[0]]
56          ELSE:
57          RETURN NULL
58          ENDIF
59
60          FUNCTION getAverage():
61          RETURN average
62          ENDFUNCTION
63
64          FUNCTION getTotal():
65          RETURN total
66          ENDFUNCTION
67
68          FUNCTION getValueAtIndex(index):
69          RETURN readings[index]
70          ENDFUNCTION
71
72          FUNCTION getValueAtTime(hours, minutes):
73          SET index = getIndexFromTime(hours, minutes)
74          RETURN getValueAtIndex(index)
75          ENDFUNCTION
76
77          FUNCTION getReadings():
78          RETURN readings[]
79          ENDFUNCTION
80
81          ENDCLASS
```

3 Algorithms

The pseudo code [2] in this section calls the methods in the *dailyReadings ADT*. This is highlighted using comments.

3.1 The maximum wind speed of a specified month and year

Input: Integer year, String month
Output: Integer windSpeed
Complexity: Constant, O(1)

3.1.1 Pseudo Code

Algorithm 3.1: Find maximum wind speed in a given month and year

```
1  FUNCTION getMaxWindSpeed(year, month):
2  SET max = 0
3
4  FOR index IN 1 TO 31:
5  SET day = READINGS[year][month][index]
6
7  IF day NOT NULL:
8  SET dayMax = day[S].getMax()
9  IF dayMax > max:
10 SET max = dayMax
11 ENDIF
12 ENDIF
13 ENDFOR
14
15 RETURN max
16
17 ENDFUNCTION
```

3.2 The median wind speed of a specified year

Input: Integer year
Output: Float median
Complexity: O(n log(n)), where n is the total number of readings per year.

3.2.1 Pseudo Code

Algorithm 3.2: Find median wind speed in a given year

```
1          FUNCTION getMedianWindSpeed(year):
2          SET windSpeedArray = []
3
4          FOR m IN 1 TO 12:
```

```
5           SET month = READINGS[year][m]
6
7           FOR d IN 1 TO 31:
8           SET day = month[d]
9
10          IF day NOT NULL:
11          // getAllReadings() returns the array of daily readings
12          JOIN windSpeedArray AND day[S].getAllReadings()
13          ENDIF
14
15          ENDFOR
16
17          ENDFOR
18
19          // Sort the windSpeedArray using QuickSort
20          SET sortedWindSpeedArray = QuickSort(windSpeedArray)
21
22          IF length of sortedWindSpeedArray is odd:
23          SET pivot = (length of sortedWindSpeedArray - 1)/2
24          SET median = sortedWindSpeedArray[pivot]
25          ELSE:
26          SET pivot = (length of sortedWindSpeedArray)/2
27          SET median = (sortedWindSpeedArray[pivot - 1] + sortedWindSpeedArray[pivot]) / 2
28          ENDIF
29
30          RETURN median
31
32          ENDFUNCTION
```

3.3 Average wind speed for each month of a specified year in the order of month

Input: Integer year
Output: Array of Integers averageWindSpeed
Complexity: Constant, O(1)

3.3.1 Pseudo Code

Algorithm 3.3: Find average wind speed per month for a given year and return values in the order of month

```
1           FUNCTION getAverageWindSpeed(year):
2           SET averageWindSpeed = []
3
4           FOR month IN (JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC):
5           SET monthAverageTotal = 0
6           SET numberOfDays = 0
7
8           FOR index 1 TO 31:
9           SET day = READINGS[year][month][index]
10
11          IF day NOT NULL:
12          SET numberOfDays = numberOfDays + 1
13          // getAverage() returns the average value for the day
14          SET monthAverageTotal = monthAverageTotal + day[S].getAverage()
```

```
15          ENDIF
16          ENDFOR
17
18          // average of averages == average (since weight is same; 288 readings per day)
19          SET monthAverage = monthAverageTotal / numberOfDays
20          PUSH monthAverage TO averageWindSpeed[]
21          ENDFOR
22
23          RETURN averageWindSpeed
24          ENDFUNCTION
```

3.4 Total solar radiation for each month of a specified year in a descending order of the solar radiation

Input: Integer year
Output: Array of Integers totalSolarRadiation
Complexity: Linear, O(n).

3.4.1 Pseudo Code

Algorithm 3.4: Return total solar radion of each month in a given year; in descending order.

```
1           FUNCTION getTotalSolarRadiation(year):
2           SET totalSolarRadiation = []
3
4           FOR month IN (JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC):
5           SET monthTotal = 0
6
7           FOR index IN 1 TO 31:
8           SET day = READINGS[year][month][index]
9
10          IF day NOT NULL:
11          // getTotal() returns the sum of readings per day
12          SET monthTotal = monthTotal + day[SR].getTotal()
13          ENDIF
14          ENDFOR
15
16          // Insert monthTotal into correct position in the array
17          FOR index IN 0 TO 11:
18          IF totalSolarRadiation[index] IS NULL:
19          SET totalSolarRadiation[index] = monthTotal
20          BREAKFOR
21          ELSEIF monthTotal > totalSolarRadiation[index]:
22          PUSH ELEMENTS FROM INDEX UPTO END ONE STEP TO THE RIGHT
23          SET totalSolarRadiation[index] = monthTotal
24          BREAKFOR
25          ENDIF
26          ENDFOR
27
28          ENDFOR
29
30          RETURN totalSolarRadiation
31          ENDFUNCTION
```

3.5 Given a date, show the times for the highest solar radiation for that date, including duplicates, displayed in reverse chronological order

Input: Integer Year, String Month, Integer Day
Output: String representation of time as HH:MM in 24 hour format
Complexity: Linear, O(n) - where n is the number of duplicates.

3.5.1 Pseudo Code

Algorithm 3.5: Show the times for the highest solar radiation for a given date, including uplicates, displayed in reverse chronological order

```
1         FUNCTION getHighestSolarRadiationTimes(year, month, day):
2
3         SET srDailyReadingsInstance = READINGS[year][month][day][SR]
4
5         // getMaxIndices() returns the array of indices of max values
6         SET maxReadingsIndices = srDailyReadingsInstance.getMaxIndices()
7
8         // getMaxIndices() method returns the array of indexes sorted in chronological order by
9         FOR index IN (length of maxReadingsIndices - 1) TO 0:
10        SET maxReadingIndex = maxReadingsIndices[index]
11
12        // getTimeFromIndex() converts the index to HH:MM string representation
13        SET time = srDailyReadingsInstance.getTimeFromIndex(maxReadingIndex)
14        PRINT time
15        ENDFOR
16
17        ENDFUNCTION
```

4 Conclusion

Most of the client requirements can be done in constant time complexity because of the way *inserts* are handled and using map ADT. The insert operation has a linear complexity O(n), where n is the number of readings. The operations like finding max, sum and average which themselves has a linear complexity are done along with the insert. Moreover, the readings are saved in a sorted array. This way, these operations are completely removed from the other algorithms which are called more frequently compared to insert. *For example, Finding the total solar radiation in a month may be called multiple times, but inserting the daily solar radiation values in only done once.* By segregating the data across various hashmaps lookups are also done in constant time rather than looping through the entire data.

4.1 Space and time requirement

4.1.1 The maximum wind speed of a specified month and year

Though the algorithm loops through the daysArray the relative time complexity can be considered constant as n here is always between 28 and 31. Space complexity is also constant as we compare the daily maximum value with overall maximum and the same variables are re-used across the loop iterations.

4.1.2 The median wind speed of a specified year

To find the median all the values needs to be sorted and saved in one single array. Combining all the readings to one array has a linear complexity O(n) where n is the number of individual arrays. However this is significantly smaller compared to the complexity of the sorting. Even though the Quick Sort complexity is O(n log(n)), n in this case can go upto 105,120 - the total readings per sensor per year. Hence the overall time complexity is O(n log(n)). This algorithm has a linear space complexity which depends on the number of readings.

4.1.3 Average wind speed for each month of a specified year in the order of month

The monthHashmap is visited in the order of the months and monthly averages are pushed into the output array in the same order. Hence the output array has been sorted in the required sequence. Since the DailyReadings class object has the average of readings as a data member, getting it has a constant complexity too. The space requirement is linear depending on the number of months with available readings.

4.1.4 Total solar radiation for each month of a specified year in a descending order of the solar radiation

Getting the total values can be done in constant complexity, however the results are sorted in place using insertion sort. Hence the algorithm has a time complexity of O(n) and space complexity of O(n).

4.1.5 Given a date, show the times for the highest solar radiation for that date, including duplicates, displayed in reverse chronological order

The algorithm loops through the *maxIndices* array, which is a data member of *DailyReadings* class. The iterations depends on the number of duplicate values and has a linear complexity, O(n). The maxIndices array is retrieved and stored

before processing and the space complexity is also linear.

References

[1] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data Structures and Algorithms in Java*. 6th ed. 2014.

[2] JD. *Pseudocode Standard*. URL: `http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html` (visited on 01/06/2018).