M22CS1.304 Data Structures and Algorithms for Problem Solving

Assignment 2

Deadline: 11:59 pm. September 19, 2022

Important Points:

1. Only C/C++ is allowed.
2. Directory Structure:

```
2021201004_A2
    |____ 2021201004_A2_Q1
    |          |____ 2021201004_A2_Q1a.cpp
    |          |____ 2021201004_A2_Q1b.cpp
    |_____ 2021201004_A2_Q2.cpp
    |_____ 2021201004_A2_Q3.cpp
    |_____ 2021201004_A2_Q4.cpp
```

   Replace your roll number in place of `2021201004`

3. Submission Format: Follow the above mentioned directory structure and zip the **`RollNo_A2`** folder and submit **`RollNo_A2.zip`** on moodle.
   Note: All submissions which are not in the specified format or submitted after the deadline will be awarded **0** in the assignment.
4. C++ STL is **not allowed** for any of the questions unless specified otherwise in the question. So "`#include <bits/stdc++.h>`" is not allowed.
5. You can ask queries by posting on the moodle.

Any case of plagiarism will lead to a 0 in the assignment or "F" in the course.
This is a hard deadline. No deadline extension requests will be entertained.

1. LRU and LFU cache

a. LRU Cache

Implement a LRU cache (Least Recently Used) Cache. It should support following functions:

- **constructor(capacity):** Creates the cache with given capacity.
- **get(key):** Return the value associated with the key, if not present return -1.
- **set(key, value):** Insert a new key value pair in the cache or update if already present. If the capacity of the cache is exceeded, then replace with the Least recently used element from the cache.

b. LFU Cache

Implement a LFU cache (Least Frequently Used) Cache. It should support following functions:

- **constructor(capacity):** Creates the cache with given capacity.
- **get(key):** Return the value associated with the key, if not present return -1.
- **set(key, value):** Insert a new key value pair in the cache or update if already present. If the capacity of the cache is exceeded, then replace it with the Least Frequently Used element. If there is tie among the Least frequent elements, choose the Least Recently Used element.

NOTE:

You are not allowed to use any STL data structures except the map/set (any of the ordered or unordered can be used).

Constraints:

1 <= Key, Value <= 10^9
1 <= Capacity <= 10^3
1 <= No. of Queries <= 10^6

2. Best Combinations

An E-Commerce Shopping website has **n** items in the inventory. Each item has a rating which can be negative also. The team wants to work on an algorithm that will suggest combinations of these items that customers might buy (or ***combos*** for short). A combo is defined as a subset of the given **n** items. The total popularity of a combo is the **sum of the popularities** of the individual items in the combo. Design an algorithm that can find the **k** combos with the highest popularity. Two combos are considered different if they have a different subset of items(i.e. combos with same popularity and different subset of items will be considered different).
Print space separated **k** integers, which is the popularity of **best k combos arranged best to worst**, where the **i^{th}** denotes the popularity of **i^{th}** best combo.
Note: You can have an empty subset as a combo as well. The popularity for such a subset is **0**.

Example:

n = 3, popularity = [3, 5, -2], k = 3

Explaination:

All possible popularitites of combos are 0, 3, 5, -2, 8, 3, 1, 6. The best three combos have popularities 8, 6, 5. The answer is [8, 6, 5]

Input Format:

- First line of input contains two space separated integers **n** and **k**.
- Second line of input contains **n** space separated integers, where the **i^{th}** integer represents the popularity of **i^{th}** item.

Output format:

- Print space seperated **k** integers denoting the popularity of best **k** combos in decreasing order of popularity.

Constratints:

1 <= n <= 10^5
-10^9 <= Popularity[i] <= 10^9
1 <= k <= min(2000, 2^n)

Expected Time Complexity:

O(n*logn + k*logk)

Sample Input:
```
4 4
1 2 3 1000
```
Sample output:
```
1006 1005 1004 1003
```

3. AVL Tree

AIM: To have end to end knowledge of the balanced binary search tree and how it can be used to solve a wide range of problems efficiently.

TASK: Implement AVL Tree with Following Operations .

	Operations	Complexity
1	Insertion	O(log N)
2	Deletion	O(log N)
3	Search	O(log N)
4	Count occurrences of element	O(log N)
5	lower_bound	O(log N)
6	upper_bound	O(log N)
7	Closest Element to some value	O(log N)
8	K-th largest element	O(logN)
9	Count the number of elements in the tree whose values fall into a given range.	O(log N)

IMPORTANT POINTS:

- Implement it with a class or struct. It should be **generic**, along with the primitive data structures(integer, float, string, etc) your code should also be able to handle class objects.
- Duplicates are allowed. (We know that AVL trees don't have duplicates but in this task you have to handle it.)

- For strings, you can simply compare them but for Class data type, you have to pass the comparator object so that you can compare two objects.
- For strings and custom Class objects, you don't need to implement the Closest Element operation.
- **Instruction for Class data type**: Name your custom comparator function as "**`cmprtr`**". Strictly follow this convention as we'll also be evaluating the program on our custom class and its custom comparator function (which will be named "**`cmprtr`**")

Tasks:

assuming the data type to be element to be **T e**, where **T** is type of element **e**

1. **insert(*e*):**
   Inserts ***e*** into the tree.

2. **delete(*e*):**
   Deletes all the occurrences of the element ***e,*** if it is present in the tree.

3. **bool search(*e*):**
   Returns **true** if ***e*** is present in the tree, otherwise returns **false**.

4. **int count_occurence(*e*):**
   Returns the count of occurrences of the element ***e***.
   Eg. If the tree has the following elements: 1, 1, 2, 2, 2, 3
   count_occurence(2) will return 3.
   count_occurence(734) will return 0.

5. **T lower_bound(e):**
   Return first element that is **greater than or equal to *e.***
   If no such element exists, return default value for type **T**.
   Eg. If the tree has the following elements: 1, 1, **2**, 2, 2, 3
   lower_bound(2) will return element corresponding to **2** i.e. 3rd element from the left.
   If the tree has the following elements 1, 1, 2, **5**, 6, 6, 7
   lower_bound(3) will return element corresponding to **5** i.e. 4th element from the left.

6. **T upper_bound(e):**
   Return the first element that is **greater than *e.***
   If no such element exists, return default value for type **T**.
   Eg. If the tree has the following elements: 1, 1, 2, 2, 2, **3**
   upper_bound(2) will return element corresponding to **3** i.e. last element from the right.
   If the tree has the following elements: 1, 1, 2, 5, 6, 6, 7
   upper_bound(7) will return **0** i.e. the default value for the type of element **e** (integer in this case)

7. **T closest_element(e):**
   Returns the element closest to ***e.***
   If no such element exists, return default value for type **T**.
   Eg. If the tree has the following elements: 1, 1, 2, 2, 2, 5.
   closest_element(2) will return 2.
   closest_element(3) will return 2.
   closest_element(4) will return 5.
   closest_element(-1472) will return 1.

8. **T Kth_largest(int k):**
   Returns the Kth largest element***.***
   If no such element exists or **k** is invalid, return default value for type **T**.
   Eg. If the tree has the following elements: 1, 1, 2, 2, 2, 3
   Kth_largest(1) will return 3.
   Kth_largest(2) will return 2.
   Kth_largest(3) will return 2.
   Kth_largest(4) will return 2.
   Kth_largest(5) will return 1.
   Kth_largest(9) will return 0 i.e. the default value for the type of element **e** (integer in this case)

9. **int count_range(T eLeft, T eRight):**
   Returns the count of the elements that lie in the range [eLeft, eRight]
   If eLeft > eRight or any other invalid case return **0**
   Eg. If the tree has the following elements: 1, 1, 2, 2, 2, 3, 4, 5, 6
   count_range(1, 6) will return 9.
   count_range(2, 3) will return 4.
   count_range(2, 2) will return 3.
   count_range(3, 10) will return 4.

Parameter to Judge: Time and space complexity.
References: AVL Tree

4. Sparse Matrix Operations

A matrix is generally represented using a 2D array. A sparse matrix is a matrix which has the majority of its elements set as 0. So using a conventional matrix representation scheme wastes a lot of memory. You are required to come up with a representation of how you will store a sparse matrix in memory using an array and linked list. And after that you are require to implement an algorithm to perform *addition*, *multiplication* and *transpose* of the matrices. For addition and multiplication operations you will be provided two sparse matrices in conventional format (conventional format is the usual representation of a 2D matrix in row x colums) and the return value should be the sum and product of the two matrices itself, respectively. Incase of transpose operation you will be provided only one sparse matrix in conventional format and you have to return its transpose.

Task:

1. **add:**
    - Takes two matrices as input (M1 and M2).
    - Performs matrix addition on these two matrices.
    - Test cases will be designed such that size of M1 = size of M2, no need to handle the cases of unequal size
    - Print the result of matrix addition.

2. **transpose:**
    - Takes one matrix as input (M1).
    - Performs matrix transpose.
    - Print the result of matrix transpose

3. **multiply:**
    - Takes two matrices as input (M1 and M2).
    - Performs matrix multiplication on these two matrices. (calculate M1 x M2)
    - Test cases will be designed such that matrix M1 and M2 are multiplication compatible.
    - Print the result of matrix multiplication.

- **Input Format:**
  - First line contains type of data structure.
    1 - Array
    2 - Linked List
  - Second line of input contains type of operation
    1 - Addition
    2 - Transpose
    3 - Multiplication
  - Third line of input contains two space separated integers N, M number of rows and columns of the first matrix respectively(note that for transpose there will be only one input matrix)
  - Following N lines will contain M space separated elements of matrix.
  - Next line will contain two space separated integers N2, M2 number of rows and columns of the second matrix respectively. (only in the case of addition and multiplication)
  - Following N2 lines will contain M2 space separated elements of matrix

- **Output Format:**
  - Print the Output matrix in conventional format (N rows, space separated M values in each row)

- **Constraints:**
  1 <= N, M <= 1000
  Number of non-zero cells <= min(N*M, 10^5)

Note:

- You are required to implement all the three operations using both the matrix representation i.e. using arrays and as well as linked lists.
- Make your code generic. The final code will be tested on all primitive data types: int, double, float, etc(excluding string). So make sure you handle these cases as well.
- Do not store the whole matrix in conventional format at any point of time(not during taking the input as well as while printing). **If found, you'll be penalized.** Take input in sparse matrix format and perform all the operations on the sparse representation of the matrix itself.