M23CS1.304 Data Structures and Algorithms for Problem Solving

Assignment 4

Deadline: 11:59 p.m. November 8th, 2023

Important Points:

1. **Only C++ is allowed**.
2. **Directory Structure:**
```
2023201001_A4
        |_____2023201001_A4_Q1.cpp
        |_____2023201001_A4_Q2.cpp
        |_____2023201001_A4_Q3.cpp
```
   Replace your roll number in place of 2023201001

3. **Submission Format:** Follow the above mentioned directory structure and zip the RollNo_A4 folder and submit RollNo_A4.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 (including vectors) 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 Moodle.

Any case of plagiarism will lead to a 0 in the assignment or "F" in the course.

1. **Treap Data Structure**

a. **Problem Statement:** Implement treap with implicit indexing.

What is a treap?

- A treap is a self-balancing data structure that combines the features of a binary search tree (BST) and a heap, using randomized priorities to maintain both ordering and prioritization of elements. It efficiently supports operations such as insertion, deletion, and search.

Underlying implementation:

- The underlying implementation of a treap relies on a randomized structure where each node has a key and priority, allowing for logarithmic time complexity operations. It does not rely on strict balance conditions but maintains balanced performance on average.

What is implicit indexing?

- In the context of a treap, implicit indexing means that elements are identified and ordered within the structure based on their positions, without explicitly assigned index values. This simplifies the treap's design and is used for efficient operations on the elements.

Operations: For **treap<T>,** where T is a generic data type, implement the following member functions. Also, print the values within **main function (driver code)** using the format mentioned for each function:

1. **bool empty()**
    - Expected Time Complexity : O(1)
    - Returns if the treap is empty or not
    - Print \`true\` or \`false\` on a new line
2. **int size()**
    - Expected Time Complexity : O(1)
    - Returns the size of the treap
    - Print the returned value on a new line
3. **void clear()**
    - Expected Time Complexity : O(n)
    - Releases all acquired memory and clears the treap
    - Don’t print anything

4. **int insert(T val)**
    - Expected Time Complexity : O(log n)
    - Inserts `val` into the treap and returns the index
    - Print the returned value on a new line
5. **bool erase(int index)**
    - Expected Time Complexity : O(log n)
    - Deletes the value at given index and returns true if the index was valid else returns false
    - Print the returned value on a new line
6. **int indexOf(T val)**
    - Expected Time Complexity : O(log n)
    - Returns the index of val if exists, otherwise returns -1
    - Print the returned value on a new line
7. **T atIndex(int index)**
    - Expected Time Complexity : O(log n)
    - Returns the value at given index if valid, otherwise returns the default value of T
    - Print the returned value on a new line (operator<< is guaranteed to work with given T)
8. **treap<T>* merge(treap<T> *t2)**
    - Expected Time Complexity : O(log n)
    - Merges t1 and t2 when called as t1->merge(t2), where t1 and t2 are treap<T> pointers. (You can assume that all the values in t1 will be less than or equal to all the values in t2).
    - Returns the pointer to the new merged treap
    - Don't print anything
9. **std::pair<treap<T>*, treap<T>*> split(int index)**
    - Expected Time Complexity : O(log n)
    - Splits the treap at given index such that the elements before the given index are in the first treap and the elements from the index(and afterwards) are in the second treap
    - Returns the pair of pointers to the new treaps
    - Don't print anything
10. **bool erase(int first, int last)**
    - Expected Time Complexity : O(log n)

    - Deletes the values between given indices (inclusive) and returns true if the whole index range was valid (completely inside the treap) else returns false
    - Print the returned value on a new line

11. **treap<T>* slice(int first, int last)**
    - Expected Time Complexity : O(log n)
    - Returns a pointer to the new treap consisting of values between given indices (inclusive) if the index range was valid else returns nullptr
    - Don't print anything
    - For testing, you need to consider the sliced treap for further commands instead of the original treap.
12. **int lower_bound(T val)**
    - Expected Time Complexity : O(log n)
    - Returns the number of elements strictly less than `val`
    - Print the returned value on a new line
13. **int upper_bound(T val)**
    - Expected Time Complexity : O(log n)
    - Returns the number of elements less than or equal to `val`
    - Print the returned value on a new line
14. **int count(T val)**
    - Expected Time Complexity : O(log n)
    - Returns the number of elements equal to `val`
    - Print the returned value on a new line
15. **std::vector<T> to_array()**
    - Expected Time Complexity : O(n)
    - Returns the vector of values in the sorted order
    - Print each element space separated

Note:

- The time complexities mentioned above are for average case and not for worst case due to the balancing mechanism to be used.
- Operations 8 and 9 will be evaluated based on output and behavior of operations 10 and 11.
- std::vector and std::pair are allowed where mentioned.

Input Format: Design an infinitely running menu-driven main function. Each time the user inputs an integer corresponding to the serial number of the operation listed above. Then, take necessary arguments related to the selected operation and execute the respective method. Finally, the program must exit with status code 0, when 0 is provided as a choice.

Output Format: As mentioned in the operation descriptions. Handle printing in the driver code and not in the library implementation.

b. **Problem Statement: Coming soon..**

2. Rope Data Structure

a. **Problem Statement:** Implement rope data structure.

What is a rope?

- A rope is a data structure used for efficiently manipulating and storing long sequences of data, such as text, in computer programs.It is a type of binary tree where each leaf (end node) holds a string and a length and each node further up the tree holds the sum of the lengths of all the leaves in its left subtree. It is designed to address the limitations of traditional strings when it comes to operations like insertion, deletion, and concatenation. The rope data structure achieves this by breaking the long sequence into smaller, more manageable chunks and organizing them in a balanced binary tree.

Underlying Implementation:

- The underlying implementation of ropes involves creating a balanced binary tree, typically as an augmented binary search tree, where each node in the tree represents a segment of the original data. This tree structure allows for efficient operations like concatenation, insertion, deletion, and substring extraction.

Operations: For **rope** implement the following member functions. Also, print the values within **main function (driver code)** using the format mentioned for each function:

1. **void rope(string s)**
    - Expected Time Complexity: O(n)
    - Constructor to create a rope for string s
2. **bool empty()**
    - Expected Time Complexity : O(1)
    - Returns if the rope is empty or not
    - Print \`true\` or \`false\` on a new line
3. **int size()**
    - Expected Time Complexity : O(1)
    - Returns the size of the rope
    - Print the returned value on a new line
4. **void clear()**
    - Expected Time Complexity : O(n)
    - Releases all acquired memory and clears the rope
    - Don't print anything
5. **boolean insert(int i, string s)**
    - Expected Time Complexity : O(log n + |s|)
    - Insert content of string s in the rope r1 from index i when called as r1->insert(i).
    - Returns true if insertion was succesful and false if index was invalid.
6. **bool erase(int first, int last)**
    - Expected Time Complexity : O(log n)
    - Deletes the values between given indices (inclusive) and returns true if the whole index range was valid (completely inside the rope) else returns false
    - Print the returned value on a new line
7. **char charAt(int index)**
    - Expected Time Complexity : O(log n)
    - Returns the value at the given index if valid, otherwise returns the null character.
    - Print the returned value on a new line.

8. **rope* subrope(int first, int last)**
    - Expected Time Complexity : O(log n)
    - Returns a new rope with elements from index first to last (inclusive).
    - Do not print anything.
9. **rope* concat(rope* r2)**
    - Expected Time Complexity : O(log n)
    - Concatenates r1 and r2 when called as r1->concat(r2), where r1 and r2 are rope pointers.
    - Returns the pointer to the new concatenated rope.
    - Don't print anything.
10. **rope* push_back(string s)**
    - Expected Time Complexity : O(log n + |s|)
    - Append content of string s in the rope r1 when called as r1->insert(i).
    - Returns a pointer to the rope with string s appended.
11. **string to_string():**
    - Expected Time Complexity : O(n)
    - Return a string denoting the content of rope r1 when called as r1->to_string().
    - Print the returned string.
12. **std::pair<rope*, rope*> split(int index)**
    - Expected Time Complexity : O(log n)
    - Splits the rope at given index such that the elements before the given index are in the first rope and the element from the index(and afterwards) are in the second rope
    - Returns the pair of pointers to the new ropes
    - Don't print anything
13. **Coming soon..**

Note : The time complexities mentioned above are for average case and not for worst case due to the balancing mechanism to be used.

Input Format: Design an infinitely running menu-driven main function. Each time the user inputs an integer corresponding to the serial number of the operation listed above. Then, take necessary arguments related to the selected operation and execute the respective method. Finally, the program must exit with status code 0, when 0 is provided as a choice.

Output Format: As mentioned in the operation descriptions. Handle printing in the driver code and not in the library implementation.

b)Problem Statement: Coming soon..

3. External Sort

Aim : External Sorting is a class of algorithms used to deal with massive amounts of data that do not fit in memory. The question aims at implementing one such type: K-Way merge sort algorithm to sort a very large array. This algorithm is a perfect example of the use of divide and conquer where with limited resources large problems are tackled by breaking the problem space into small computable subspaces and then operations are done on them.

Task : Given a file containing a large unsorted list of integers (Will not fit in your usual Laptop RAM) as input, generate an output file with non-descending sorted list of given integers.

Input format :

- Input.txt contains space separated integers.
- Program should take two command line arguments.
- First command line argument is the input file path.
- Second command line argument is the output file path.
- For example: You have a directory named data in your current working directory inside which you have kept the input.txt file and you wish to create output.txt inside the data directory itself.Then, run the program using the following command.

  ```
  ./a.out ./data/input.txt ./data/output.txt
  ```
- Input files can be arbitrarily large.

Output Format :

- Print the following details in the terminal window:
  - Number of integers in a temporary file.

    - Number of temporary files created.
    - Total time taken by the program up to 2 decimal places.
- Output file should contain integers in non-descending sorted order in a new line.

- **Evaluation Parameters:**
    - Correctness.
    - Time and space complexity of the algorithm.
    - Efficient use of data structures.

- **Generation of unsorted file**
    - To generate the unsorted file, a python script is uploaded along with this pdf. It contains all the instructions required to run it.

- **Note:** You are allowed to use **vector** and inbuilt **sort** for this problem