National University of Computer & Emerging Sciences, Karachi
Fall-2018CS-Department
Midterm II (Sol)
November 12, 2018 Slot:9AM – 10AM

Course Code:CS201	**Course Name: Data Structures**
Instructor Name: Muhammad Rafi / Nida Pervaiz	
Student Roll No:	**Section No:**

- Return the question paper.
- Read each question completely before answering it. There are **3 questions and 2 pages.**
- In case of any ambiguity, you may make assumption. But your assumption should not contradict with any statement in the question paper.
- All the answers must be solved according to the sequence given in the question paper.
- Be specific, to the point while coding, logic should be properly commented, and illustrate with diagram where necessary.

Time: 60 minutes. **Max Marks**: 50 points

Sorting Algorithm
Question No. 1 **[Time: 20 Min] [Marks: 5+15]**

a. Outline some key differences in Quick sort and Merge sort.

Key Differences Between Quick Sort and Merge Sort

1. In the merge sort, the array must be parted into just two halves (i.e. n/2). As against, in quick sort, there is no compulsion of dividing the list into equal elements.
2. The worst case complexity of quick sort is $O(n^2)$ as it takes a lot more comparisons in the worst condition. In contrast, merge sort have the same worst case and average case complexities, that is O(n log n).
3. Merge sort can operate well on any type of data sets whether it is large or small. On the contrary, the quick sort cannot work well with large datasets.
4. Quick sort is faster than merge sort in some cases such as for small data sets.
5. Merge sort requires additional memory space to store the auxiliary arrays. On the other hand, the quick sort doesn't require much space for extra storage.
6. Merge sort is more efficient than quick sort.
7. The quick sort is internal sorting method where the data that is to be sorted is adjusted at a time in main memory. Conversely, the merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in the auxiliary memory.

b. Time intervals can be represented by a pair of integers (a,b), this suggest that interval start from time *a* and ends at time *b* both inclusive. Two intervals (a,b) and (c,d) are overlapping if the ends time of first is greater than the start time of the second interval. For a concrete example (2,5) and (4,7) are overlapping. We can merge these two intervals to create a single combined interval for example a combined interval will be (2,7). One FASTIAN suggests a very indigenous approach for merging a set of intervals. He suggested the use of a single stack of interval type. From the set of intervals, he first sorts all interval on start time using any efficient sorting algorithm like (merge sort or heap sort), he pushes the first interval (from the sorted intervals) on an empty stack, while from the second interval of the set, he only checks if the top of the interval on stack is overlapping, then he merges the two interval as per the large ending time, and push back the merged interval on stack. While completely processing the set of intervals in this way, the resultant stack now contains all merged intervals. Write a function to implement the approach for merging a set of intervals.
Example:
Set of Intervals = {(2,4), (3,7), (8,12), (10,15), (17, 18)}
Merges set of non-overlapping intervals = {(2,7), (8,15), (17,18)}

Pseudocode

Define a class for storing and manipulating Intervals. It will have two integers for time intervals. Some member's functions for getter and setter along with constructors.

Read all the intervals using an DynamicSafeArray<Interval> Intervals [20]

Sort all intervals on increasing order of start time.

Define a Stack of Interval type, Stack<Interval>. Push the first interval of sorted array to stack.

In a loop for each interval in Intervals array.

a. Pick the current interval, check if the current interval is not overlapped with top of the stack interval, put it on stack.
b. if the current interval is overlapped with the top interval of the stack. merged the two intervals, by updating the end-time of the stack interval with the end-time of the current interval.

In the end the stack contains all non-overlapped merged intervals.

Stack, Queue and Priority Queue	
Question No. 2	**[Time: 20 Min] [Marks: 20]**

One FASTIAN suggested a very crafty implementation of combined Stack and Queue in an array. He used a DynamicSafeArray suggested during the course to use for it. He called it StackQ. The idea is to keep a queue (FIFO) from the right side of the array, while a stack (LIFO) from the left side of the array. The class implementation for this StackQ is having three indexes, one for top of the stack and two for both front and rear of the queue. The following diagram represents one instance of this data structures at some point in time.

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
12	76	43					15	8	7
		top ^					**rear^**		**front^**

Here is the class definition for some primitive functions for combined stack and queue.

```
template<class T>
class StackQ{

      private:
      DynamicSafeArray<T> *Data;
       unsigned int top; unsigned int front; unsigned int rear;

      public:

      //Stack Primitive
      void Push(T element){
         if (top+1! =rear)
         {top++;
          data[top]=element;
         }}
      void Pop(){ if (top > -1) top--;}
      T Peek(){ if (top != -1) return Data[top];}
      bool IsfullStack(){return ((top+1==rear)||(top==size-1));}
      bool isEmptyStack(){ return (top == -1);}
      //Queue primitive
      bool IsFullQueue(){ return ((rear==top+1)||(rear==0));}
      bool IsEmptyQueue(){ return (front == rear = size);}
      void Enqueue(T element){
       if((rear-1!=top) || (rear!=0)){
              rear--;
             data[rear]=element;}}
      void Dequeue(){ if ((front < size)&&(front >-1)) front--;}
      T Process(){ return Data[front];}

};
```

Provide some valid implementation of all the primitives of stack and queue in this scenario. Your code must check all necessary conditions to support any primitive operation.

Hashing & Searching	
Question No. 3	**[Time: 15 Min] [Marks: 5+5]**

a. Given an array A of n >= 10000 distinct positive integers. Write an algorithm that will output one element x of A such that x is not among the top 5 elements of A, neither is it among the bottom 5 elements of A. Note that the top and bottom 5 elements of A are the first 5 and the last 5 elements when A is sorted. Your algorithm should not take more than say 50 comparisons. Only comparisons count! All other arithmetic/memory operations are free.

A tricky approach would be to randomly pick eleven elements from the array (as all are distinct positive integers). The median of these eleven will surely be greater than 5 smallest elements and less than 5 biggest elements of the given n. Use either heap-sort or merge sort to sort and pick the median element. This is the required element. These sort obviously do much less than 50 comparisons. Another tricky approach will be use quick sort partition logic, recursively, to pick the sixth largest element from the randomly picked 11 elements of the array.

b. Define how hashing perform search. What is so special about hashing?

Hashing is a mathematical transformation of the data into a key for constant time lookup for these data. The hashing approach guarantee to search the data in a pre-allocated memory (Hash-Table) with data in O (1) time. It computes an index from the data and directly access the data at the index location. In search computing Hashing is a perfect choice for O(1) time lookup/search.