National University of Computer & Emerging Sciences, Karachi
Fall-2020 CS-Department
Final Examination (Sol)
January 01, 2021 Timing: 8:30 AM – 11:30 AM

Course Code: CS 218	Course Name: Data Structures
Instructor Name:	Muhammad Rafi / Dr. Ghufran/Basit Ali/ Zain ul Hassan
Student Roll No:	Section No:

- Return the question paper.
- Read each question completely before answering it. There are **7 questions and 5 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: 180 minutes. **Max Marks**: 100 points

Object Oriented Programming / Dynamic Safe Arrays	
Question No. 1	**[Time: 15 Min] [Marks: 10]**

Consider the class skeleton given below. You are required to implement all the functions given in the Bold Font.

```
Date(const int a[]) {
   DateData= new int [3];
       for( int i=0; i < 3; i++)
      {
         *(DateData+i)=*(a+i);
       }
}

Date(const Date & rhs) {
   DateData= new int [3];
  for( int i=0; i < 3; i++)
   { *(DateData+i)=    *(rhs.DateData+i); }
}

~ Date() {
     if(DateData != 0)
   {
    delete [] DateData;
    }
}

Date * operator=( const Date & rhs){
   if (this != &rhs)
  { delete [] this->DateData;
     DateData= new int [3];
     for( int i=0; i < 3; i++)
     {
      *(DateData+i)=    *(rhs.DateData+i);
      }
    }
  return(*this);
}
```

Linked List and Variants	
Question No. 2	**[Time: 20 Min] [Marks: 15]**

a. One FASTIAN developed a very different solution to represent polynomial for computer manipulation. He has used a Linked List to hold each term of the polynomial for example:

List1: $3X^8+2X^6-2X^5+4X^4+2X^3+3$
List2: $2X^8$
Result: $5X^8+2X^6-2X^5+4X^4+2X^3+3$

For the above List1 that contains six nodes representing six terms, the first node contains attributes as Co-efficient=+3, Base=X and Exponent=8, and in the same fashion he has stored different number of terms in two different instances of lists. The polynomial linked list is always stored in descending order of power and in single variable (Base). You are required to write a program that can add two instances of this lists and store its result in another list. [5]

```
SinglyLinkedList& addPolynomials(SinglyLinkedList & L1,
SinglyLinkedList & L2)
{
 SinglyLinkedList<Terms>  NewPolynomial= New
 SinglyLinkedList<Terms>();

 SinglyLinkedNode<Terms> * CurrentL1 = L1.head;
 SinglyLinkedNode<Terms> * CurrentL2 = L2.head;

while (CurrentL1 != 0 && CurrentL2 !=0)
{
  if( CurrentL1->Exponent  == CurrentL2 ->Exponent)
  {
  NewPolynomial.AddAtTail(CurrentL1->Coefficient+CurrentL2->
  Coefficient, CurrentL1->Base, CurrentL1 ->Exponent);
  CurrentL1 = CurrentL1->Next;
  CurrentL2 = CurrentL2->Next;
  }
 Else if (CurrentL1->Exponent  >  CurrentL2 ->Exponent)
 {
    NewPolynomial.AddAtTail(CurrentL1->Coefficient, CurrentL1->Base,
  CurrentL1 ->Exponent);
  CurrentL1 = CurrentL1->Next;
 }
 Else
 {
  NewPolynomial.AddAtTail(CurrentL2->Coefficient, CurrentL2->Base,
  CurrentL2 ->Exponent);
  CurrentL2 = CurrentL2->Next;
 }
}

While(CurrentL1){
  NewPolynomial.AddAtTail(CurrentL1->Coefficient, CurrentL1->Base,
  CurrentL1 ->Exponent);
  CurrentL1 = CurrentL1->Next; }

While(CurrentL2){
  NewPolynomial.AddAtTail(CurrentL2->Coefficient, CurrentL2->Base,
  CurrentL2 ->Exponent);
  CurrentL2 = CurrentL2->Next; }}
```

b. Write a function for DoublyLinkedList<T> which decide if the two pointers to the nodes of the given list are adjacent or not? [5]

```
bool IsAdjacent(DNode<T> * left, DNode<T> * right){

if((left->next == right && right->prev==left) ||
(left->prev=right && right->next ==left))
  return (TRUE);
else
return (FALSE);
}
```

c. Write a function for ReverseSinglyLinkedList<T> which reverse a given singly linked list in its vanilla implementation (only holding a head pointer to access the list). [5]

```
SinglyLinkedList<T>* ReverseSinglyLinkedList(Node<T> * Head)
    {
        Node<T>* current = head;
        Node<T> *prev = NULL, *next = NULL;

        while (current != NULL) {
            next = current->next;
            current->next = prev;
            prev = current;
            current = next;
        }
        head = prev;

       SinglyLinkedList<T>  ReverseList = new
     SinglyLinkedList<T>(Head)

      return (ReverseList);
 }
```

Stack, Queues, Heaps and Priority Queues	
Question No. 3	**[Time: 25 Min] [Marks: 15]**

a. You are given Queue data structure that supports standard operations like enqueue(), dequeue(), etc. You need to implement a Stack data structure using only instances of Queue and queue operations allowed on those instances. Illustrate how your Stack perform operations like Push, Pop and Peek. Also give pseudo-code for these functions. [5]

```
class StackQ {
    // Two inbuilt queues
    queue<int> q1, q2;
   // To maintain current number of
    // elements
    int curr_size;

public:
    StackQ()
    {    curr_size = 0;     }

    void push(int x)
    {
        curr_size++;
          // Push x first in empty q2
        q2.push(x);
          // Push all the remaining
        // elements in q1 to q2.
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }
          // swap the names of two queues
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void pop()
    {
        // if no elements are there in q1
        if (q1.empty())
            return;
        q1.pop();
        curr_size--;
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }
};
```

b. 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. [10]

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:

      StackQ();
      StackQ(int size);
      StackQ(constStackQ&rhs);
      StackQ operator=(constStackQ&rhs);
      ~StackQ();
      //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.

Recursion	
Question No. 4	**[Time: 15 Min] [Marks: 10]**

a. Suggest a recursive version of Bubble Sort. Identify the base case and recursive case. [5]

A recursive version of Bubble sort can easily be defined with two different approaches. First if we push the heavy-weight element from the array to the last index, our problem is reduced from size n to n-1, as we have already placed it to the final position. Similarly, the lowest bubble.
Base case: if we only have one element in the array just return it.
Recursive case: If there are more than two elements in the array, place the heaviest at the last index. This reduce the problem eventually into base case.

b. Give the pseudo-code/algorithm for the recursive Bubble Sort and identify it running time. [5]

```
void bubbleSort(int arr[], int n)
{
   // Base case
   if (n == 1)
      return;

   // One pass of bubble sort. After
   // this pass, the largest element
   // is moved (or bubbled) to end.
   for (int i=0; i<n-1; i++)
      if (arr[i] > arr[i+1])
         swap(arr[i], arr[i+1]);

   // Largest element is fixed,
   // recur for remaining array
   bubbleSort(arr, n-1);
}
// The running time of this approach will be n^2.
```

Sorting, Hashing and Searching	
Question No. 5	**[Time: 15 Min] [Marks: 10]**

a. What is a stable sort? Is merge sort stable? Can we make an unstable sorting algorithm stable? How? [2]

Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list. Merge sort is stable.
An unstable sorting algorithm can be made stable by transforming it to another collection (say B) with ordered pair(A[i],i) whcih keep relative position for the given array. Hence B = (A[i],i). Sort the array B with equal keys keeping position relative. Take all relative values of B-sorted into Sorted-A that is first value of each pair.

b. Consider a hash table of size 10 with hash function h(k) = k^2 mod 10. Draw the table that results after inserting, in the given order, the following values: 19, 26, 13, 48, 17, 6, 9, 20, 3, 1 for each of the following scenarios below:

i. When collisions are handled by separate chaining. [2]

1. 19^2 mod 10 = 1
2. 26^2 mod 10 =6
3. 13^2 mod 10= 9
4. 48^2 mod 10 =4
5. 17^2 mod 10 =9
6. 6^2 mod 10 =6
7. 9^2 mod 10 =1
8. 20^2 mod 10 =0
9. 3^2 mod 10 =9
10. 1^2 mod 10=1

Hash(Index)	KeyPointers
0	20
1	19,9,1
2	
3	
4	48
5	
6	26,6
7	
8	
9	13,17,3

ii. When collisions are handled by linear probing. [2]

1. 19^2 mod 10 = 1
2. 26^2 mod 10 =6
3. 13^2 mod 10= 9
4. 48^2 mod 10 =4
5. 17^2 mod 10 =9
6. 6^2 mod 10 =6
7. 9^2 mod 10 =1
8. 20^2 mod 10 =0
9. 3^2 mod 10 =9
10. 1^2 mod 10=1

Hash(Index)	Key
0	20
1	19
2	9
3	1
4	48
5	
6	26
7	6
8	
9	13

- Could not insert **17** as no space for probing
- Could not insert **3** as no space for probing

iii. When collisions are handled by double hashing using a second hash function h'(k) = 5 – (k mod 5). [2]

1. 19^2 mod 10 = 1
2. 26^2 mod 10 =6
3. 13^2 mod 10= 9
4. 48^2 mod 10 =4
5. 17^2 mod 10 =9
   h`(k) = 5-(17 mod 5)=1
6. 6^2 mod 10 =6
   h`(k) = 5-(6 mod 5)=4
7. 9^2 mod 10 =1
   h`(k) = 5-(9 mod 5)=1
8. 20^2 mod 10 =0
9. 3^2 mod 10 =9
   h`(k) = 5-(3 mod 5)=2
10. 1^2 mod 10=1
    h`(k) = 5-(1 mod 5)=4

Hash(Index)	Key
0	20
1	19
2	9
3	
4	48
5	1
6	26
7	
8	
9	13

- Could not place 17 as no space
- Could not place 6 as no space after probing
- Could not place 3 as no space after probing

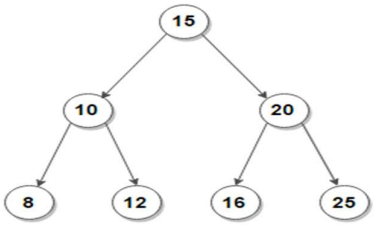
c. Devise an algorithm for Ternary Search. Unlike binary search it divide the collection into 3 equal portions and omit 2/3 portion of the collation during the search. Give the pseudo-code for the same.

```
int ternarySearch(int array[], int start, int end, int key) {
   if(start <= end) {
      int midFirst = (start + (end - start) /3);
        //mid of first and second block
      int midSecond = (midFirst + (end - start) /3);
        //mid of first and second block
      if(array[midFirst] == key)
         return midFirst;
      if(array[midSecond] == key)
         return midSecond;
      // where the key in the sub-part of the array
      if(key < array[midFirst])
         return ternarySearch(array, start, midFirst-1, key);
      if(key > array[midSecond])
         return ternarySearch(array, midSecond+1, end, key);
      return ternarySearch(array, midFirst+1, midSecond-1, key);
   }
   return -1;
}
```

[2]

Trees and Variants	
Question No. 6	**[Time: 40 Min] [Marks: 20]**

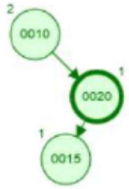
a. Given a BST with its root pointer and a value X surely presented in the given tree. You need to write a routine to find the next larger key (large than X) from the tree. For example: in the given BST if X=10 then 12 will be the answer and if X=12 then 15 will be the answer. [5]



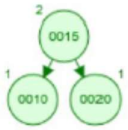

```
BTNode<T> * FindNextHigher(BTNode<T> * root, int X) {
        BTNode<T> * next = null;
        BTNode<T>* curr = root;
        while (curr != null) {
            if (curr->Data > X) {
                next = curr;
                curr = curr->left;
            } else {
                curr = curr->right;
            }
        }
        return next;
    }
}
```

b. Insert the following sequence of elements into an AVL tree, starting with an empty tree: 10, 20, 15, 25, 30, 16, 18, 19. From the resultant AVL tree try to delete 30 and give the final resultant tree. Show each step and a brief description about it. [5]

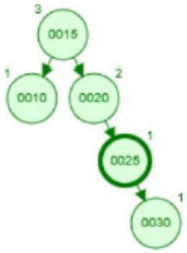
Inserting 10 and 20 is simple as there is no imbalance for height. Inserting 15 will create an imbalance tree:



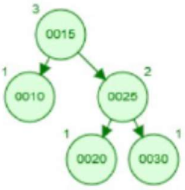
Doing a double left rotation, we get



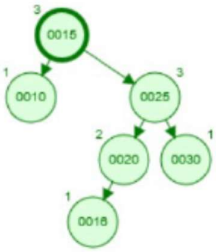
Inserting 25 is ok, inserting 30 next make the tree imbalance.



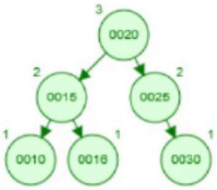
Doing a single left rotation make it balance again.

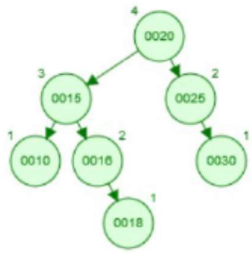
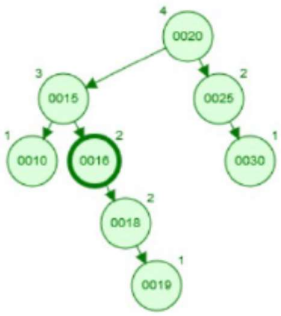


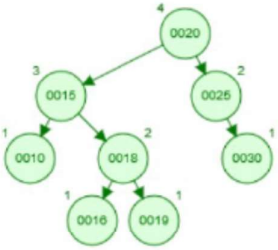
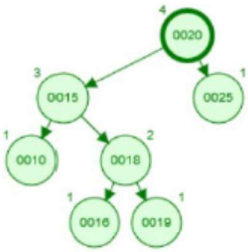
Inserting 16 again make the AVL tree imbalance

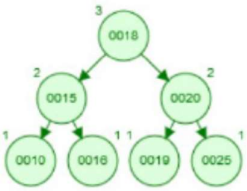


Doing a double rotation left gives us:



Inserting 18 does not do any harm for AVL	Inserting 19 will again make AVL Tree imbalance
	

Doing single left rotation fix it again.	Deleting 30 will make an imbalance tree
	

Doing a double rotation right will give us:	
	

c. Given a sorted array of n positive integer write a function that create a balance binary tree with best execution time using the following function definition. [5]

```
template<class T>
BTNode<T>*  BST<T>::CreateBalanceBST(T data[], int first, int last, BTNode<T>* root)
{
if (first <= last) {
int middle = (first + last)/2;
root.insert(data[middle]);
CreateBalanceBST (data,first,middle-1,root);
CreateBalanceBST (data,middle+1,last,root);
}
return root;
}
```

d. Write a recursive function that takes a Binary Search Trees (BST) root pointer and return height of the given tree. [5]

int HeightOfTheTree(const BTNode<T> *tree)

```
Height of a Binary Tree:  The height for a null tree is 0,
which the height of a binary tree is the distance from root
to the farthest leaf node.

int HeightOfTheTree(BTNode* root)
{
    // Base case: empty tree has height 0
    if (root == 0)
        return 0;

 // recur for left and right subtree and consider farthest
//each call add one  to the height
    return 1 + max(HeightOfTheTree(root->left),
HeightOfTheTree(root->right));
}
```

Graphs	
Question No. 7	**[Time: 40Min] [Marks: 20]**

a. Define Minimum Spanning Tree (MST). Illustrate at least 3 differences between Prim's and Kruskal's algorithm for finding Minimum Spanning Tree. [5]

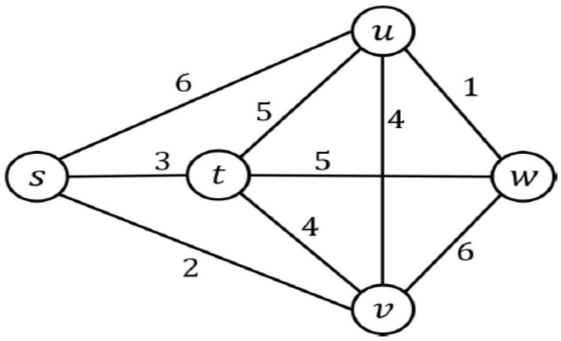
Kruskal's Algorithm	**Prim's Algorithm**
Kruskal's Algorithm will grow a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest. It can grow many trees simultaneously, so it considers a forest of trees for a valid solution.	Prim's Algorithm will grow a solution from a random vertex by adding the next cheapest vertex to the existing tree. Prims algorithm grows a single tree.
Kruskal's Algorithm is faster for sparse graphs.	Prim's Algorithm is faster for dense graphs.
O(E log E)	O(E log V)

b. Using the data structures known to devise an algorithm that when given a G (V, E) in any valid representation finds whether the graph is a single connected component or a disjoint graph. [5]

Let's consider the graph G (V, E) is given in adjacency matrix form, we can apply any BFS or DFS to the graph by considering any vertex as starting point. The solution is true for directed graph as well. Similarly, you can use Union-Find data structures to see if all the vertices are part of a single set.

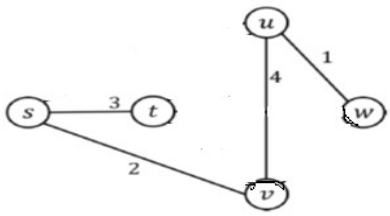
```
void dfs (int v){
    if vis[v] is true:
        return
    set vis[v] as true
    for all neighbors u of v:
        if vis[u] is false:
            dfs(u)
}

bool IsConnected(){
    for all nodes u:
        dfs(u);
        if vis[u] is false:
            return false
    return true
}
```

c. Consider the graph given below:



i. Draw the minimum spanning tree that would result from running Prim's algorithm on this graph, starting at vertex s. List the order in which edges are added to the tree. [5]

Starting from s it will first select vertex (v) from edge (s,v), then it will select vertex (t) from edge(s,t), it will then select vertex (u) from edge(v,u) and finally it will select vertex (w) edge(u,w) hence complete it spanning to the entire graph.



ii. Draw the minimum spanning tree that would result from running Kruskal's algorithm on this graph. List the order in which edges are added to the tree. [5]

The Kruskal's method first select edge (u,w) as it is the smallest edge. It will then select edge (s,v), it will then select edge(s,t) so far it does not form any cycle. The next it will pick either edge(t,v) or (u,v) as both have the same value 4, if it will pick edge (t,v) it will form a cycle and the edge will be rejected because it is the heaviest edge in the cycle. It will eventually pick edge (u,v) and hence get the minimum spanning tree.

<The End.>