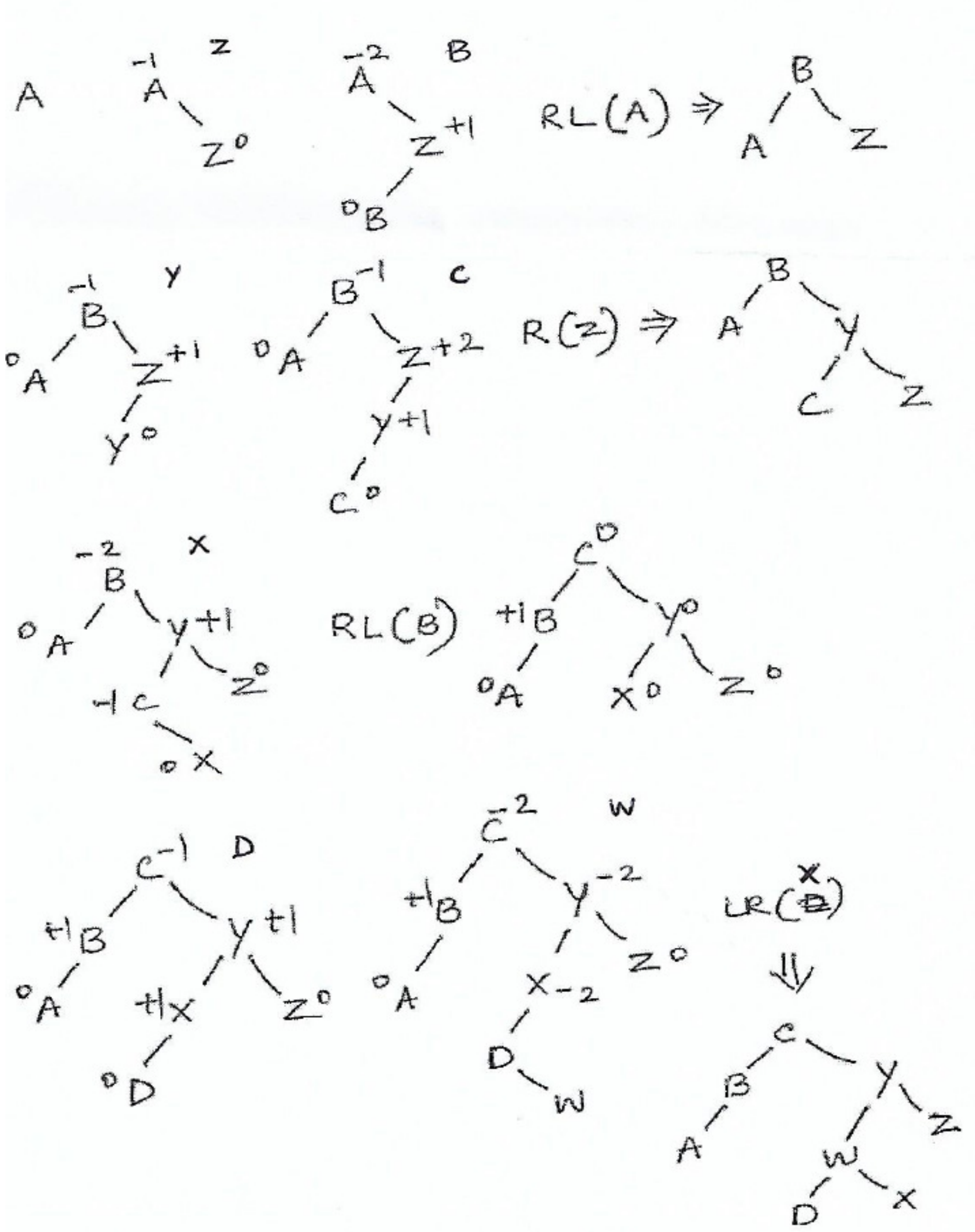
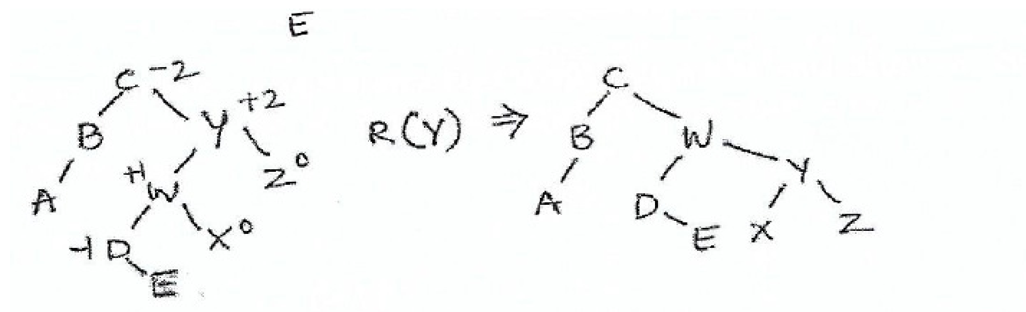
		PES University, Bengaluru (Established under Karnataka Act No. 16 of 2013)	**UE19CS202**

NOVEMBER 2020: IN SEMESTER ASSESSMENT B Tech III SEMESTER
TEST – 2

UE19CS202 – DATA STRUCTURES AND ITS APPLICATIONS

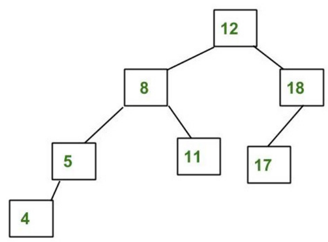
Scheme and Solution

1.	a)	What is an AVL Tree? An AVL Tree is a binary search tree where in the balance factor of every node in the tree is either -1, 0 or +1. The Balance factor of a node is the difference between heights of the left and the right subtree of that node	2
	b)	Insert the following keys in the order shown, to build them into an AVL Tree A , Z, B, Y, C, X, D, W. 	6

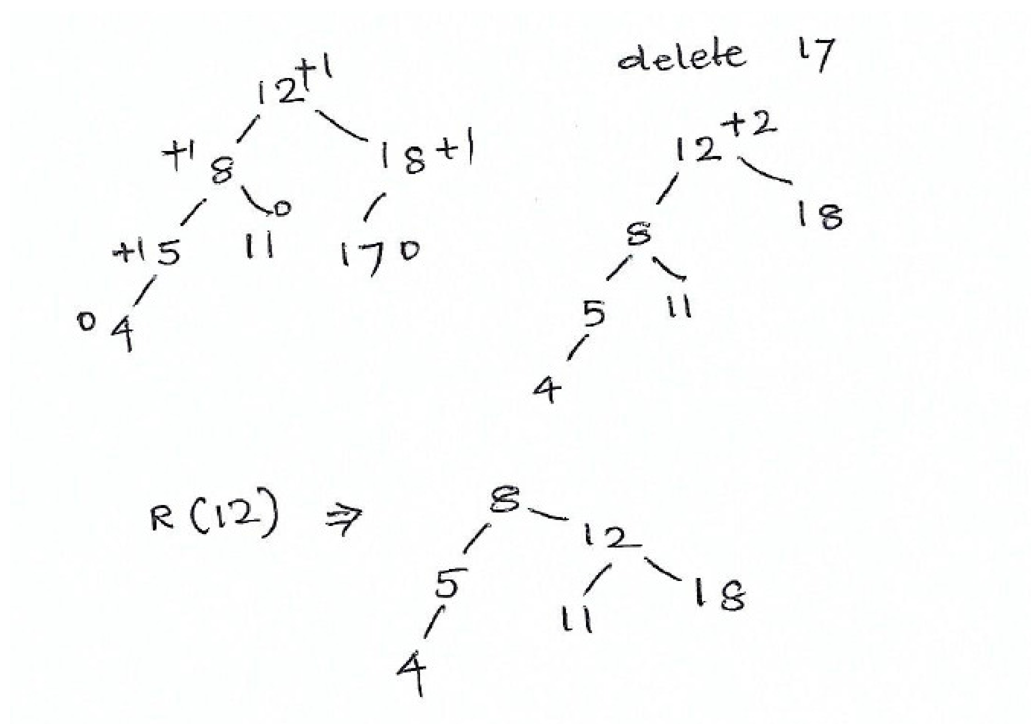


There are five rotations performed .One of the Double Rotations(except the first one)can be given 2 marks, others carry 1 mark each.

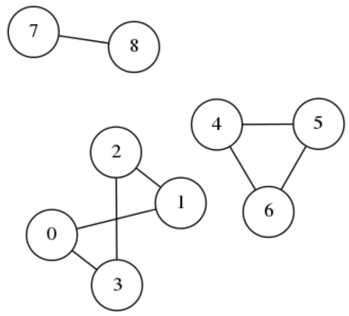
c) 2



Re Create the above AVL Tree after deleting the key 17. **(2 marks)**



2\. a) Given an undirected graph represented as adjacency matrix, write a function using DFS to find the number of connected components. You need to write the DFS function also — 6



For example , There are three connected components in the above graph

int component() - **3 marks**

```
{
        int i,j;
        label=0;

        for(i=1;i<=n;i++)
        {
          if(visit[i]==0)
         {
            ++label;
            dfs(i);
               }
        }
        return label;
}
```

void dfs(int v**) 3 marks**

```
{
        int u;
        visit[v]=label;

        for(u=1;u<=n;u++)
        {
          if((a[v][u]==1)&&(visit[u]==0))
              dfs(u);
        }
}
```

b) Write a function to implement the breadth first search traversal for the graph represented as an adjacency list. You need not write the queue functions. — 4

```
void bfs(int v)
 {
          int w, j;
         struct node *p;
         visit[v]=1;
         printf("%d",v);
         qinsert(v);

         while(!qisempty())
         {
```

		`w=qdelete();`
		
		`for(p=a[w];p!=NULL;p=p->next)`
		`{`
		`j=p->data;`
		`if(visit[j]==0)//not visited`
		`{`
		`visit[j]=1;`
		`printf("%d ",j);`
		`qinsert(j);`
		`}`
		`}`
		`}`
		`}`	

3.	a)	Write a function to insert words consisting of characters of lower case into a trie. Consider the following structure of the trie node
		struct trienode
		{
		struct trienode *child[26];
		int endofword;
		}
		Function prototype : insert(struct trienode *root, char *key).
		
		void insert(struct trienode *root, char *key) **4 marks**
		{
		struct trienode *curr;
		int i,index;
		
		curr=root;
		for(i=0;key[i]!='\0';i++)
		{
		index=key[i]-‘a’;
		if(curr->child[index]==NULL)
		curr->child[index]=getnode();
		curr=curr->child[index];
		}
		curr->endofword=1;
		}
		
		struct trienode *getnode() **2 marks**
		{
		int i;
		struct trienode *temp;
		temp=(struct trienode*)malloc(sizeof(struct trienode));
			6

```
  for(i=0;i<26;i++)
      temp->child[i]=NULL;
   temp->endofword=0;
  return temp;
}
```

b) Write a function to display the words stored in a Trie in the lexicographic order. Use the same structure of the Trie node defined in the above question — **4**

```
void display(struct trienode *curr)
 {
   int i,j;
   for(i=0;i<26;i++)
   {
    if(curr->child[i]!=NULL)
    {
      word[length++]=i+'a';
      if(curr->child[i]->endofword==1)//if end of word
      {
        printf("\n");
        for(j=0;j<length;j++)
          printf("%c",word[j]);
      }
     display(curr->child[i]);
    }
   }
   length--;
   return ;
 }
```

---

4. a) Explain the following — **4**

a)Quadratic probing

b) Double Hashing

Quadratic probing is an open-addressing scheme where we look for i^2'th slot in i'th iteration if the given hash value x collides in the hash table.

Let hash(x) be the slot index computed using the hash function.

- If the slot hash(x) % S is full, then we try (hash(x) + 1*1) % S.
- If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S.
- If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S.
- This process is repeated for all the values of i until an empty slot is found.

(2 marks)

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

(hash1(key) + i * hash2(key)) % TABLE_SIZE

		Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table. **(2 marks)**	
	b)	Suppose a hash table contains HASHSIZE =13 entries indexed from 0 to 12 and that the following keys are to be mapped into the table. 10,100,32,45,58,126,3,29,200,400 Determine the hash addresses and find how many collisions occur when the hash function **key % HASHSIZE** is used. Show the contents of the hash table when the collision is resolved using linear probing.	6

Computation of Hash addresses

1) 10 : 10 % 13 = 10
2) 100 : 100 %13 = 9
3) 32 : 32 % 13 = 6
4) 45 : 45 % 13 = 6 (collision)
5) 58 : 58 % 13 = 6 (collision)
6) 126 : 126 % 13 = 9 (collision)
7) 3 : 3 % 13 = 3
8) 29 : 29 % 13 = 3 (collision)
9) 200 : 200 % 13 = 5
10) 400 : 400 % 13 =10 (collision)

No of Collisions : 5

Hash Table Contents after application of Linear Probing

			3	29	200	32	45	58	100	10	126	400
0	1	2	3	4	5	6	7	8	9	10	11	12

3 marks for computing the hash addresses
1 mark for the collisions
2 marks for the hash table contents after applying linear probing

Note: The functions written above are one of the ways of implementation. The same functions may be written using different function prototypes.