SRN | | | | | | | | | | | | |

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

UE17CS202

Dec 2018: END SEMESTER ASSESSMENT (ESA) B.TECH.

UE17CS202- Data Structures Scheme and Solution

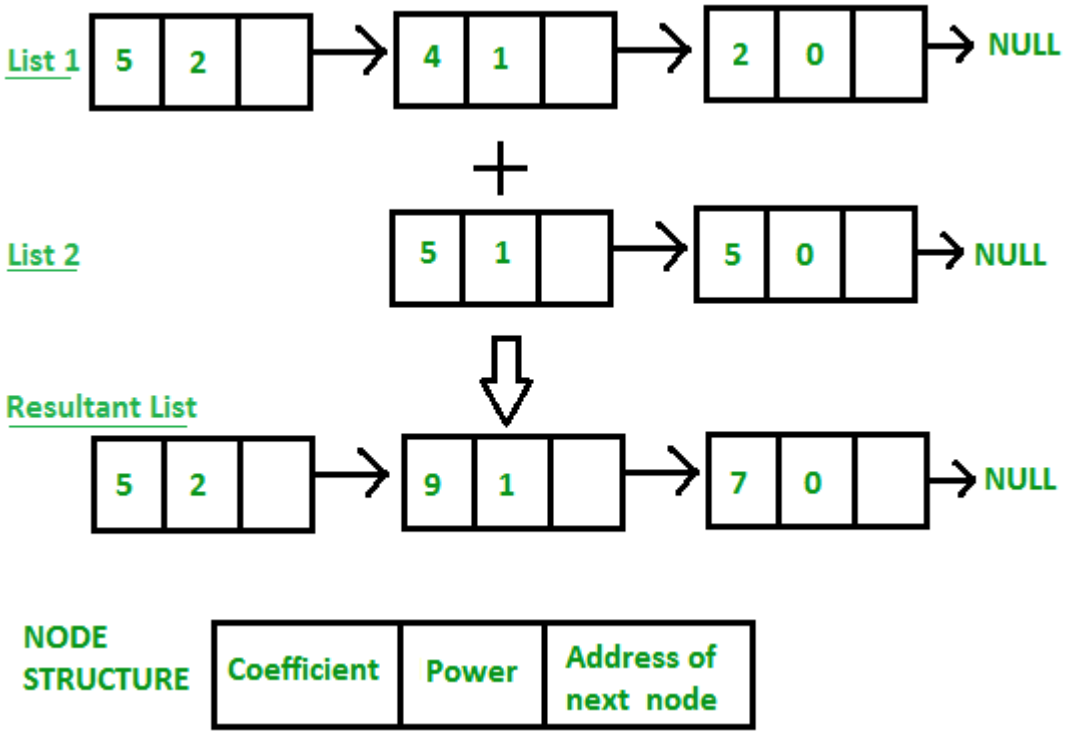
Time: 3 Hrs | Answer all questions preferably in the same order | Max Marks: 100

NOTE: **Detailed algorithm or C code is acceptable. Show all steps and State any assumptions made.**

1	a)	Give 2 reasons of preferring an array over a list. Similarly, give 2 reasons when list is preferred over an array? Random access is faster in array than in linked list. Less space taken in array since no pointers. Insertion and deletion of elements is faster in Linked list. Linked list is used when the number of elements is not known ahead and is variable.	2 2

	b)	Explain how polynomial arithmetic of single variable can be implemented using singly linked list. (i) Specify structure of each node of the list.(ii) Show with an example of how polynomial addition and multiplication operations can be performed using linked list.
		
		// Node structure containing power and coefficient of variable
		struct Node
		{
		int coeff;
		int pow;
		struct Node *next;
		};
		
		Input:
		1st number = 5x^2 + 4x^1 + 2x^0
		2nd number = 5x^1 + 5x^0
		Addition Output:
		5x^2 + 9x^1 + 7x^0
		Multiplication Output:
		25x^3 + 45x^2 + 30x^1 + 10x^0
		
		Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.
		
		Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result.
			2
			
			2
			
			2

SRN | | | | | | | | | | | | |



c) Specify the node structure of an integer doubly-linked list (dll). Write a function to delete a node based on the specified index position in the dll. (0 should delete the head node, 1 should delete node after the head and so on). Make sure you handle the boundary conditions.

2

```
struct dnode
{
        int data;
        struct dnode *llink;
        struct dnode *rlink;
};
typedef struct dnode d_node;
```

1

```
struct list
{
        d_node* head;
        d_node* tail;
        int number_of_nodes;
};
typedef struct list d_list;
```

2

```
int delete_At_Position(d_list* ptr_list, int index)
{
        d_node* curr = ptr_list -> head;
        if(index < 0 || index > (ptr_list -> number_of_nodes - 1) || ptr_list ->
number_of_nodes == 0)
        {
            return -1;
        }
        for(int i = 0; i < index; ++i)
        {
            curr = curr -> rlink;
```

```
        }
        d_node* prev = curr -> llink;
        d_node* next = curr -> rlink;
        if(prev != NULL)
        {
            prev -> rlink = next;
        }
        else
        {
            ptr_list -> head = next;
        }
        if(next != NULL)
        {
            next -> llink = prev;
        }
        else
        {
            ptr_list -> tail = prev;
        }
        free(curr);
        --ptr_list -> number_of_nodes;
}
```

3

2. a) Convert the following infix expression to its equivalent postfix and prefix expressions. Write the equivalent binary expression tree.

A * (B + D) / E – F * (G + H / K)

Postfix : ABD+*E/FGHK/+*- — 2
Prefix : -/*A+BDE*F+G/HK — 2
Binary expression tree (last page) — 2

b) Write a method that will take two sorted stacks A and B (min on top) and create one stack that is sorted (min on top). You are allowed to use only the stack operations such as pop, push, empty, size and top. Other than stacks, no other data structures such as arrays are allowed.

```
Stack A, B, C, D;

while (!empty(A) && !empty(B))
{
   if (top(A) < top(B))
      push(C, pop(A));
   else
      push(C, pop(B));
}
if (empty(A)
   while (!empty(B))
      push(C, pop(B));

if (empty(B)
   while (!empty(A))
      push(C, pop(A));
```

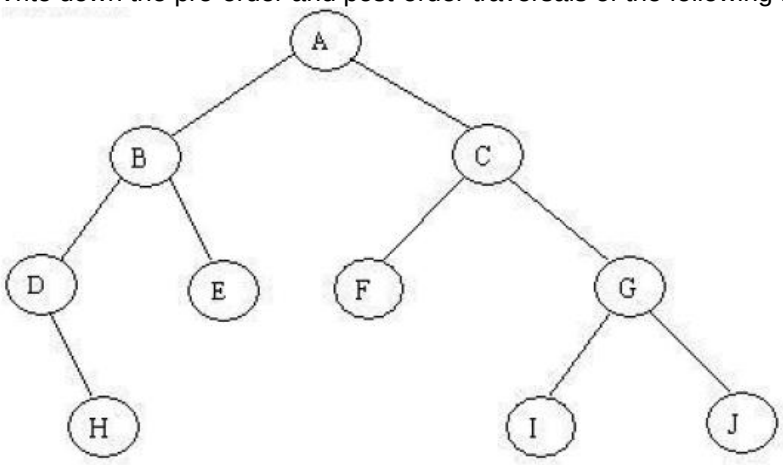
3

1

1

		while (!empty(C))
		push(D, pop(C));
		return (D);
			1

	c)	Implement enqueue and dequeue operations of a circular integer queue using an array of size N. Assume f is the index to the front element of the queue and r is index where the next element gets inserted into the queue. Implement any other auxiliary function that may be needed.
		
		int size() { return (N – f + r)modN }
		int empty() { return (size() == 0) }
		
		int dequeue()
		{ if empty() then “Queue Empty”
		int data = q[f]
		f =(f +1) mod N;
		return data
		}
		
		void enqueue(int e):
		{ if size() = = N-1 then “Queue Full”
		Q[r]= e; r=(r+1) mod N;
		}
			1
			
			3
			
			1
			3

3.	a)	Write down the pre-order and post-order traversals of the following binary tree.  •Preorder: A B D H E C F G I J •Postorder: H D E B F I J G C A	2 2

	b)	Write a function that can find the number of connected components in a graph using depth first search (DFS). Use the Adjacency Matrix representation of graph.
		
		int a[20][20],reach[20],n;
		
		void dfs(int v)
		{
		int i;
		reach[v]=1;
		for(i=1;i<=n;i++)
		if(a[v][i] && !reach[i])
		dfs(i);
		}
			8

```
void main()
{
 int i,j,count=0;
 clrscr();
 printf("n Enter number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
  reach[i]=0;
  for(j=1;j<=n;j++)
   a[i][j]=0;
 }
 printf("n Enter the adjacency matrix:n");
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   scanf("%d",&a[i][j]);

 dfs(1);
 printf("n");
 for(i=1;i<=n;i++)
  if(!reach[i])
   {
        dfs(i);
        count++;
    }
  printf("The Graph has %d  connected components", count);
 }
```

c) Given a binary tree and a sum, return 1 if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return 0 if no such path can be found. (For example, in the binary tree in question 3(a), A+B+D+H is one such root-to-leaf path)

```
int hasPathSum(struct tnode* node, int sum)

// Strategy: subtract the node value from the sum when recurring down,
// and check to see if the sum is 0 when you run out of tree.

int hasPathSum(struct node* node, int sum) {
    // return true if we run out of tree and sum==0
    if (node == NULL) {
        return(sum == 0);
     }
    else {
        // otherwise check both subtrees
        int subSum = sum - node->data;
        return(hasPathSum(node->left, subSum) ||  hasPathSum(node->right, subSum));
    } // if
}
```

2

3

3

4. a) What data structures can be used to implement a priority queue?. Explain why a heap is preferable to other data structures to implement a priority queue?

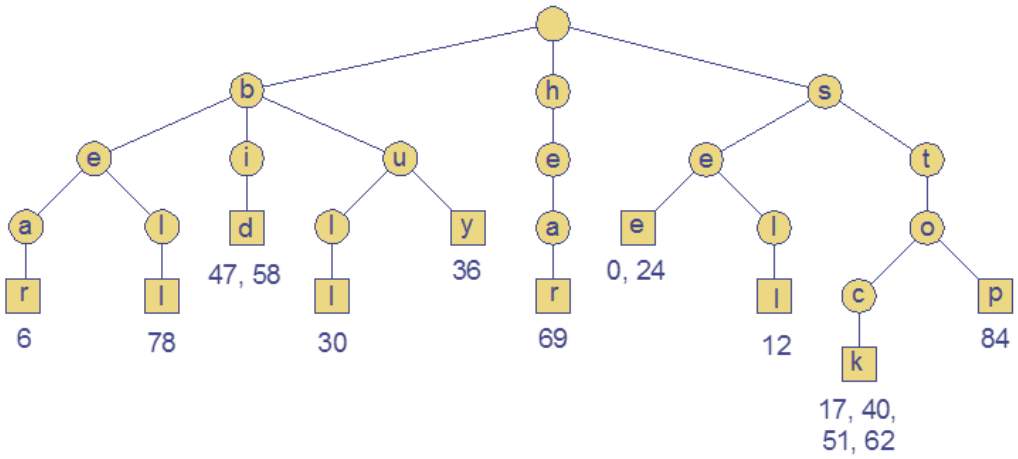
		Arrays and Linked list can be used to implement priority queues. The elements can be kept in sorted order during insertion and the first element which has the highest priority can be returned. However, insertion can be slow as the number of elements increases. Heap is a complete binary tree where the root element has the highest priority. Building a heap, insertion and deletion all are proportional to the height of the tree which is much faster than the priority queue implementation using arrays and linked list.	2 2
	b)`	Draw a Binary Search Tree with these numbers showing the intermediate trees. 44, 17, 88, 32, 28, 65, 54, 29, 82, 76, 97 (i). Redraw after adding 80 and 35. (ii) Redraw after 44 is removed from the BST.	8
	c)	A priority queue is implemented using max heap where the maximum element is at the root of the heap. Assume the integer array *h[ ]* stores the heap elements and *count* stores the number of heap elements. Implement a function called RemoveMax() to remove the maximum element of the heap and adjust to form a heap again. int RemoveMax(int *h, int *count); // returns the maximum element of heap h and heapify.	8

```
int RemoveMax(int *h, int *count)
  {
   int t;
   t=h[0];
   h[0]=h[*count-1];
   (*count)--;
   adjust(h,*count); //re-create the heap
   return t;
  }

 void adjust(int *h, int count)
  {
   int i,j,key;

   j=0;
   key=h[j];
   i=2*j+1;//get the left child

   while(i<=count-1)//as long as the left child exists
   {
    if((i+1)<=count-1)//check whether right child exists
      if(h[i+1]>h[i])//get the largest child
        i++;
    if(key<h[i])
    {
      h[j]=h[i];//move the child up
      j=i;
      i=2*j+1;//get the new child position
     }
    else
     break;
   }
   h[j]=key;
```

		}	
5.	a)	How can a trie tree be used to find the position of word occurrences in a page of text?. Draw suitable diagrams to illustrate your solution. (eg. all positions of a word “stock” in a page of text) Assume the page of text is stored in a character array where each word now has a starting index position in the array. Insert each word into a trie tree. At the leaf node (when word ends) store the index position of the word in the array as follows. When a word is matched, check whether the word ends in a leaf node. The leaf node contains all the position of the word occurences. 	6

	b)	Suggest a hash function that hashes a string of characters to an index of a hashtable of size TSIZE. Two words of same characters (top, pot) should hash to different indices.
		Any function that incorporates the position of the character in the string. Here each character is multiplied by a power of 37 being a prime number.
		$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize-i-1] \cdot 37^i$
		
		int hash (const string &key, int tableSize)
		{
		int hashVal = 0;
		
		for (int i = 0; i < key.length(); i++)
		hashVal = 37 * hashVal + key[i];
		
		hashVal %= TSIZE;
		if (hashVal < 0) /* in case overflows occurs */
		hashVal += TSIZE;
		
		return hashVal;
		};
			6

	c)	Draw the 11-entry hash that results from using the hash function h(i) = (2i+5) mod 11 to hash keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5. Show the hash table when collisions are handled by using (i) Chaining (ii) Linear Probing.	8

i	12	44	13	88	23	94	11	39	20	16	5
2*i+5	29	93	31	181	51	193	27	83	45	37	15
Mod11	7	5	9	5	6	6	5	6	1	4	4

	Chaining				Linear Probing
0				0	11
1	20			1	39
2				2	20
3				3	5
4	16, 5			4	16
5	44, 88, 11			5	44
6	94, 39			6	88
7	12, 23			7	12
8				8	23
9	13			9	13
10				10	94