Q.P.22955 (1)

Solution (Answer Key)

Q1. A) A **data structure** is a specialized format for organizing and storing **data**. an **abstract data type** (**ADT**) is a mathematical model for **data types**, where a **data type** is **defined** by its behavior (semantics) from the point of view of a user of the **data**, specifically in terms of possible values, possible operations on **data** of this **type**, and the behavior of these operations.

B) **Asymptotic Notations** are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.

C) A **recursive function** (**DEF**) is a **function** which either calls itself or is in a potential cycle of **function** calls.

D) A **stack** is a basic **data structure** that can be logically thought as linear **structure** represented by a real physical **stack** or pile, a **structure** where insertion and deletion of items takes place at one end called top of the **stack**. ... They are 1) inserting an item into a**stack** (push). 2) deleting an item from the **stack** (pop).

E) A **red-black tree** is a balanced binary search **tree** with the following **properties**: Every node is colored **red** or **black**. ... If a node is **red**, then both its children are **black**. Every simple path from a node to a descendant leaf contains the same number of **black** nodes.

F) A set of items connected by edges. Each item is called a vertex or node. Formally, a **graph** is a set of vertices and a binary relation between vertices, adjacency.

G) A **linked list** is a linear collection of **data** elements, in which linear order is not given by their physical placement in memory. Instead, each element points to the next. It is a **data structure** consisting of a group of nodes which together represent a sequence.

```
1. Q2. A) #include <stdio.h>
2.
3. #define MAX 50
4. int queue_array[MAX];
5. int rear = - 1;
6. int front = - 1;
7. main()
8. {
9.      int choice;
10.     while (1)
11.     {
12.         printf("1.Insert element to queue \n");
13.         printf("2.Delete element from queue \n");
14.         printf("3.Display all elements of queue \n");
15.         printf("4.Quit \n");
16.         printf("Enter your choice : ");
17.         scanf("%d", &choice);
18.         switch (choice)
19.         {
20.             case 1:
21.             insert();
22.             break;
23.             case 2:
24.             delete();
25.             break;
```

```
26.             case 3:
27.             display();
28.             break;
29.             case 4:
30.             exit(1);
31.             default:
32.             printf("Wrong choice \n");
33.         } /*End of switch*/
34.     } /*End of while*/
35. } /*End of main()*/
36. insert()
37. {
38.     int add_item;
39.     if (rear == MAX - 1)
40.     printf("Queue Overflow \n");
41.     else
42.     {
43.         if (front == - 1)
44.         /*If queue is initially empty */
45.         front = 0;
46.         printf("Inset the element in queue : ");
47.         scanf("%d", &add_item);
48.         rear = rear + 1;
49.         queue_array[rear] = add_item;
50.     }
51. } /*End of insert()*/
52.
53. delete()
54. {
55.     if (front == - 1 || front > rear)
56.     {
57.         printf("Queue Underflow \n");
58.         return ;
59.     }
60.     else
61.     {
62.         printf("Element deleted from queue is : %d\n", queue_array[front]);
63.         front = front + 1;
64.     }
65. } /*End of delete() */
66. display()
67. {
68.     int i;
69.     if (front == - 1)
70.         printf("Queue is empty \n");
71.     else
72.     {
73.         printf("Queue is : \n");
```

Answer Key 55

```
74.         for (i = front; i <= rear; i++)
75.             printf("%d ", queue_array[i]);
76.         printf("\n");
77.     }
78. } /*End of display() */
```

B) deletion operation in a binary heap

Deleting an Element from the Heap

- Deletion always occurs at the root of the heap.
- If we delete the root element it creates a hole or vacant space at the root position.
- Because the heap must be complete, we fill the hole with the last element of the heap.
- Although the heap becomes complete, i.e. it satisfies the shape property, the order property of heaps is violated.
- As the value that comes from the bottom is small, we have to perform another operation to satisfy the order property.
- This operation involves moving the element down from the root position until either it ends up in a position where the order property is satisfied or it hits a leaf node.

Q3. A) Merge sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */

quickSort(arr[], low, high)
```

Answer key 33

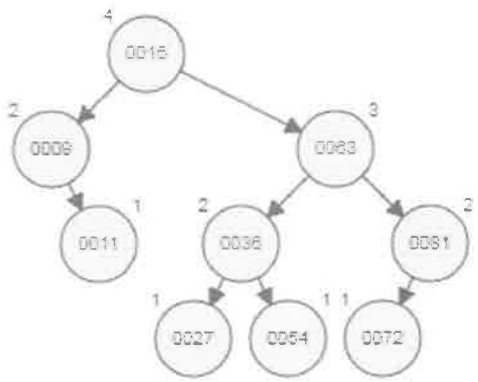
4

```
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

B) AVL tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.



Q4. A) Singly Linked List.

```
#include<iostream>
using namespace std;

/* defines the structure of a single linked list node*/
typedef struct list_node {
  int data;
```

```
  struct list_node *next; // pointer to next node in the list
}node;

/* create new node */
node *getNewNode(int data) {
  node *new_node = new node;
  new_node->data = data;
  new_node->next = NULL;
  return new_node;
}

/* displays the list elements */
void displayList(node *head) {
  cout << "Displaying List : ";
  while (head != NULL) {
    cout << head->data << " -> ";
    head = head->next;
  }
  cout << "NULL " << endl;
}

/* Search the node with element as data
  Return the pointer to the node if found else return NULL */
node *searchNode(node *head, int data) {
  node *ptr = NULL;
  while (head) {
    if (head->data == data) {
      ptr = head;
      break;
    }
    head = head->next;
```

Answer Key 55

```
  }
  return ptr;
}

/* insert a node at the beginning of the list */
node *insertNodeBeg(node *head, int data) {
  node *ptr = getNewNode(data);
  if (head == NULL) { // if list is empty
    head = ptr;
  }
  else {
    ptr->next = head;
    head = ptr;
  }
  return head;
}

/* insert a node at the end of the list */
node *insertNodeEnd(node *head, int data) {
  node *ptr = getNewNode(data);
  if (head == NULL) { //if list is empty
    head = ptr;
  }
  else {
    node *temp = head;
    while (temp->next != NULL) { // move to the last node
      temp = temp->next;
    }
    temp->next = ptr; // insert node at the end
  }
  return head;
```

Answer Key 55

```
}

/* insert a node at the after a particular node in the list */
node *insertNodeAfter(node *head, int element, int data) {
  // search the element after which node is to be inserted
  node *temp = searchNode(head, element);
  if (temp == NULL) { // element not found
    cout << "Element not found ... " << endl;
  }
  else {
    node *ptr = getNewNode(data);
    if (temp->next == NULL) { // node has to inserted after the last node
      temp->next = ptr;
    }
    else {  // insert the node after the first or an intermediate node
      ptr->next = temp->next;
      temp->next = ptr;
    }
  }
  return head;
}

/* delete a particular node from the list */
node *deleteNode(node *head, int element) {
  node *temp = searchNode(head, element); // search the node to be deleted
  if (temp == NULL) { // element not found
    cout << "Node to be deleted not found ... " << endl;
  }
  else {
    if (temp == head) { // first node is to be deleted
      head = head->next;
```

Answer Key 55

```
      delete temp;
    }
    else { // node to deleted is an intermediate or last node
      node *ptr = head;
      while (ptr->next != temp) {
        ptr = ptr->next;
      }
      ptr->next = temp->next;
      delete temp;
    }
  }
  return head;
}

int main() {
  node *head = NULL;
  head = insertNodeBeg(head, 7);      // 7
  head = insertNodeBeg(head, 9);      // 9 -> 7
  head = insertNodeEnd(head, 11);     // 9 -> 7 -> 11
  head = insertNodeAfter(head, 9, 4);  // 9 -> 4 -> 7 -> 11
  head = insertNodeAfter(head, 7, 3);  // 9 -> 4 -> 7 -> 3 -> 11
  head = insertNodeAfter(head, 11, 6); // 9 -> 4 -> 7 -> 3 -> 11 -> 6
  displayList(head);
  head = deleteNode(head, 7);         // 9 -> 4 -> 3 -> 11 -> 6
  head = deleteNode(head, 6);         // 9 -> 4 -> 3 -> 11
  head = deleteNode(head, 9);         // 4 -> 3 -> 11
  displayList(head);
  return 0;
}.
```

B)Stack.

```
#include <stdio.h>
```

Answer Key 55

```c
#include <stdlib.h>

struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;

int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();

int count = 0;

void main()
{
    int no, ch, e;

    printf("\n 1 - Push");
    printf("\n 2 - Pop");
    printf("\n 3 - Top");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Dipslay");
    printf("\n 7 - Stack Count");
    printf("\n 8 - Destroy stack");

    create();

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
        case 2:
            pop();
            break;
        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {
                e = topelement();
                printf("\n Top element : %d", e);
            }
```

Answer Key 55

```
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            stack count();
            break;
        case 8:
            destroy();
            break;
        default :
            printf(" Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* Create empty stack */
void create()
{
    top = NULL;
}

/* Count stack elements */
void stack_count()
{
    printf("\n No. of elements in stack : %d", count);
}

/* Push data into stack */
void push(int data)
{
    if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
    count++;
}

/* Display stack elements */
void display()
{
    top1 = top;
```

```
    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }

    while (top1 != NULL)
    {
        printf("%d ", top1->info);
        top1 = top1->ptr;
    }
}

/* Pop Operation on stack */
void pop()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("\n Error : Trying to pop from empty stack");
        return;
    }
    else
        top1 = top1->ptr;
    printf("\n Popped value : %d", top->info);
    free(top);
    top = top1;
    count--;
}

/* Return top element */
int topelement()
{
    return(top->info);
}

/* Check if stack is empty or not */
void empty()
{
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements", count);
}

/* Destroy entire stack */
void destroy()
{
    top1 = top;

    while (top1 != NULL)
    {
        top1 = top->ptr;
        free(top);
        top = top1;
```

Answer Key 55

```
        top1 = top1->ptr;
    }
    free(top1);
    top = NULL;

    printf("\n All stack elements destroyed");
    count = 0;
}
```

Q5. A) A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. In detail explain the example.

Prims Algorithm

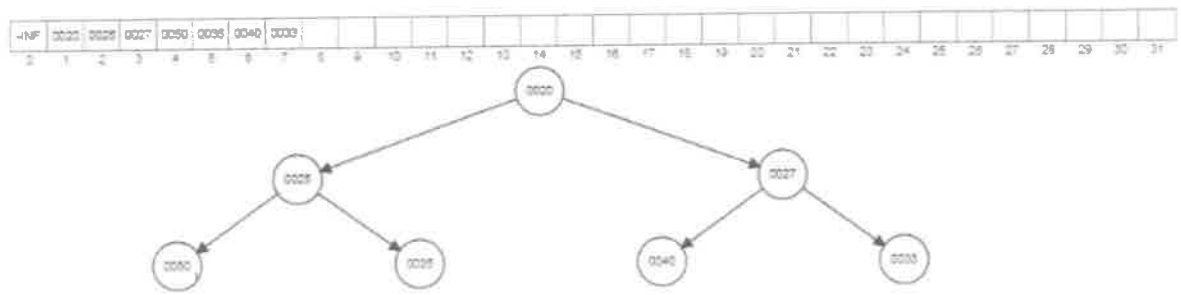
	node 1	node 2	node 3	node 4	node 5	node 6	node 7
node 1		3	6		8		
node 2				7		3	
node 3	6						
node 4		7	1				
node 5	8			2		5	
node 6					5		2
node 7					3		

Run Prim!

The weight of the minimum spanning tree is 14.
The arcs used are highlighted above in red.

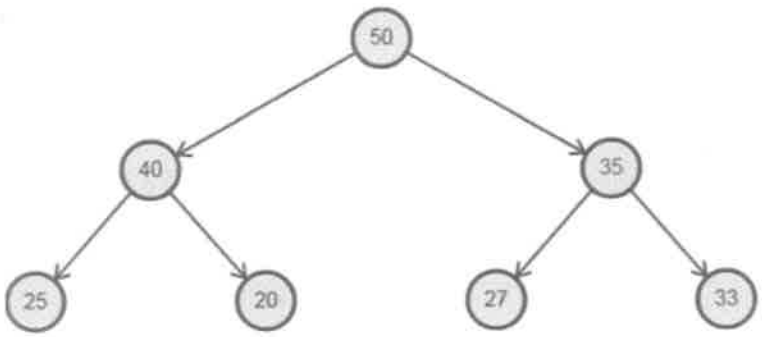
Kruskals Algorithm-Answer will be same.

Q.5 B)Min Heap



Max Heap

Answer Key 55



Q.6 a)Euclid's Algorithm

The Euclidean Algorithm is a technique for quickly finding the GCD of two integers. ***The Algorithm***

The Euclidean Algorithm for finding GCD(A,B) is as follows:

- If A = 0 then GCD(A,B)=B, since the GCD(0,B)=B, and we can stop.
- If B = 0 then GCD(A,B)=A, since the GCD(A,0)=A, and we can stop.
- Write A in quotient remainder form (A = B·Q + R)
- Find GCD(B,R) using the Euclidean Algorithm since GCD(A,B) = GCD(B,R)

Q.6 b) Huffman tree

The algorithm is as:
List all the letters used, including the "space" character, along with the frequency with which they occur in the message.
Consider each of these (character, frequency) pairs to be nodes; they are actually leaf nodes, as we will see.
Pick the two nodes with the lowest frequency. If there is a tie, pick randomly amongst those with equal frequencies.
Make a new node out of these two, and turn two nodes into its children.
This new node is assigned the sum of the frequencies of its children.
Continue the process of combining the two nodes of lowest frequency till the time only one node, the root is left.

Q.6 c) Sparse matrix

In numerical analysis and computer science, a sparse matrix or sparse array is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense. The number of zero-valued elements divided by the total number of elements (e.g., m × n for an m × n matrix) is called the sparsity of the matrix (which is equal to 1 minus the density of the matrix).

Q.6 d) Breadth First Search Algorithm

```
BFS (G, s)
//Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s )
 //Inserting s in queue until all its neighbour vertices are marked.
```

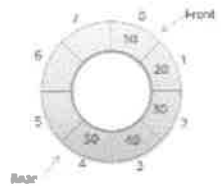
Answer Key 55

```
mark s as visited.
while ( Q is not empty)
  //Removing that vertex from queue,whose neighbour will be visited now
     v  =  Q.dequeue( )

    //processing all the neighbours of v
    for all neighbours w of v in Graph G
         if w is not visited
                  Q.enqueue( w )
   //Stores w in Q to further visit its neighbour
                  mark w as visited.
```

Q.6 e) Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



Q.6 f) Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.