SET B – Yellow Color Key

Part A

1. d
2. b
3. a
4. c
5. b
6. b
7. c
8. c

Part B

9. Write a C program to reverse a given string using stack.

- Create an empty stack.
- One by one push all characters of string to stack.
- One by one pop all characters from stack and put them back to string.

```
// C program to reverse a string using stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    char* array;
};

// function to create a stack of given
// capacity. It initializes size of stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack
        = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array
        = (char*)malloc(stack->capacity * sizeof(char));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
```

```
    return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

// Function to add an item to stack.
// It increases top by 1
void push(struct Stack* stack, char item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}

// Function to remove an item from stack.
// It decreases top by 1
char pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // Create a stack of capacity
    // equal to length of string
    int n = strlen(str);
    struct Stack* stack = createStack(n);

    // Push all characters of string to stack
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);

    // Pop all characters of string and
    // put them back to str
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

// Driver program to test above functions
int main()
```

```
{
    char str[] = "Welcome";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}
```

10. BT Search

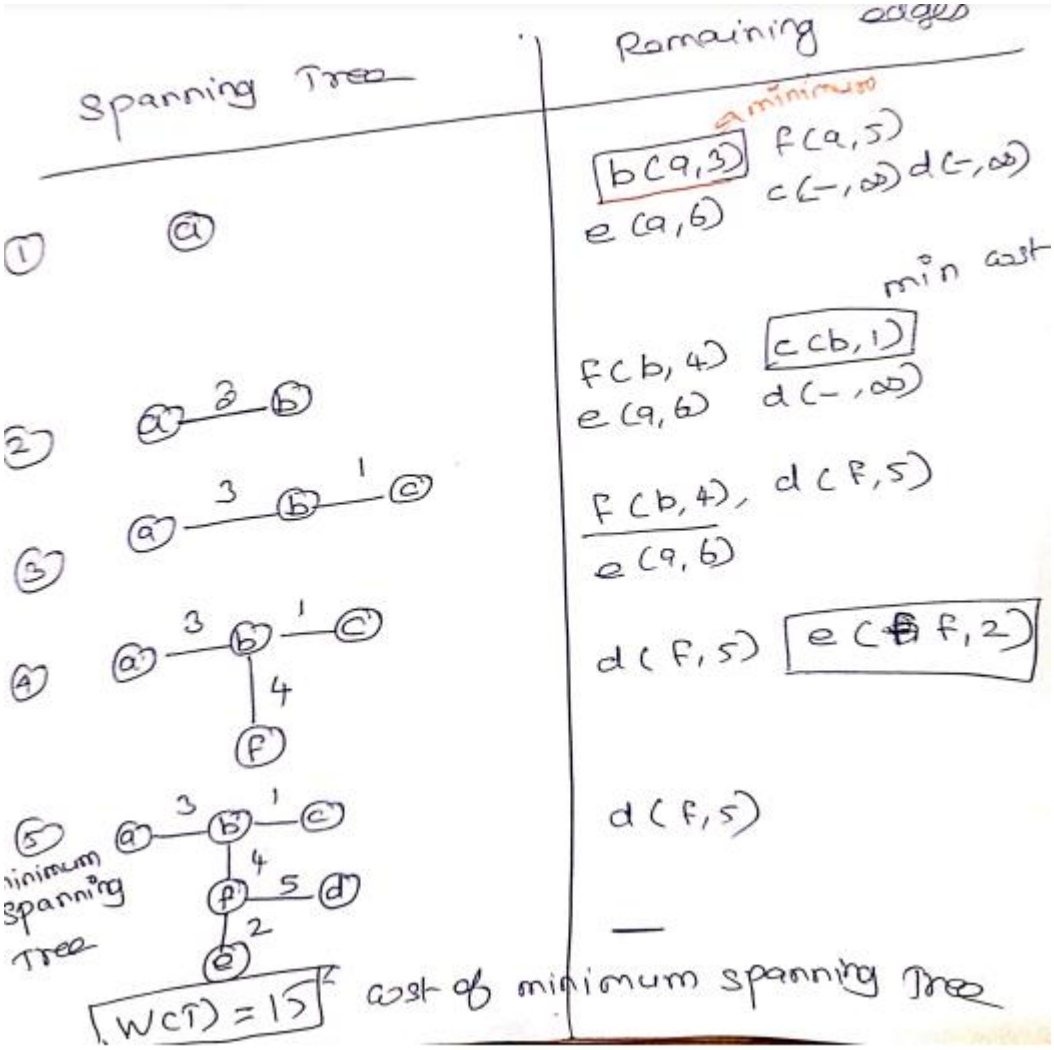
```
struct node {
   int key;
   struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item)
{
   struct node* temp
      = (struct node*)malloc(sizeof(struct node));
   temp->key = item;
   temp->left = temp->right = NULL;
   return temp;
}
struct node* search(struct node* root, int key)
{
   // Base Cases: root is null or key is present at root
   if (root == NULL || root->key == key)
      return root;

   // Key is greater than root's key
   if (root->key < key)
      return search(root->right, key);

   // Key is smaller than root's key
   return search(root->left, key);
}
```

11. Prims Algorithm for finding minimum spanning tree



PART C

12. A)

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
    - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '**(**'], then push it in the stack. ['^' operator is right associative and other operators like '+','–','*' and '/' are left-associative].
        - Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.

- In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Ex: 623+-382/+*2^3+

Incoming Character	Contents of Stack	Value
6	6	
2	6, 2	
3	6, 2, 3	
+	6, 5	2 + 3 = 5
-	1	6 – 5 = 1
3	1, 3	
8	1, 3, 8	
2	1, 3, 8, 2	
/	1, 3, 4	8 / 2 = 4
+	1, 7	3 + 4 = 7
*	7	1 * 7 = 7
2	7, 2	
^	49	7 ^ 2 = 49

12.A) B

START

Procedure Hanoi(disk, source, dest, aux)

```
IF disk == 1, THEN
  move disk from source to dest
ELSE
  Hanoi(disk - 1, source, aux, dest)    // Step 1
  move disk from source to dest         // Step 2
  Hanoi(disk - 1, aux, dest, source)    // Step 3
```

END IF

END Procedure
STOP

12.b

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full

```
void enqueue(struct Queue* queue, int item)

{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

int dequeue(struct Queue* queue)
{
    if (isEmpty(queue)) {
        printf("\nQueue is empty\n");
        return;
    }
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
bool isEmpty(struct Queue* queue)
{
    return (queue->size == 0);
}
bool isFull(struct Queue* queue)
{
    return (queue->size == queue->capacity);
}
```

13. a. Max Heap

13.b. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

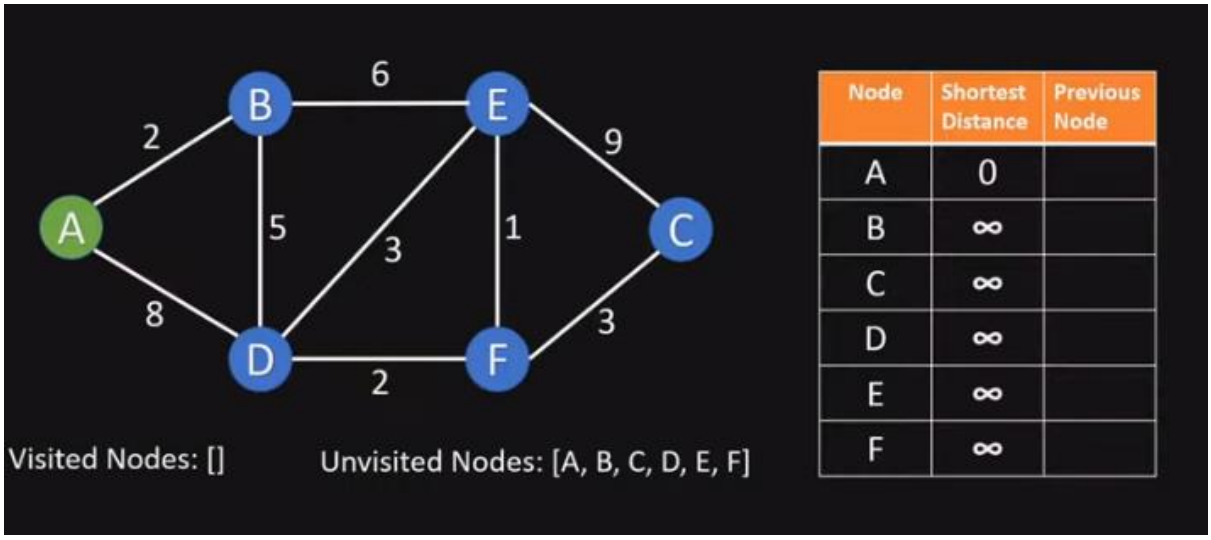
(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

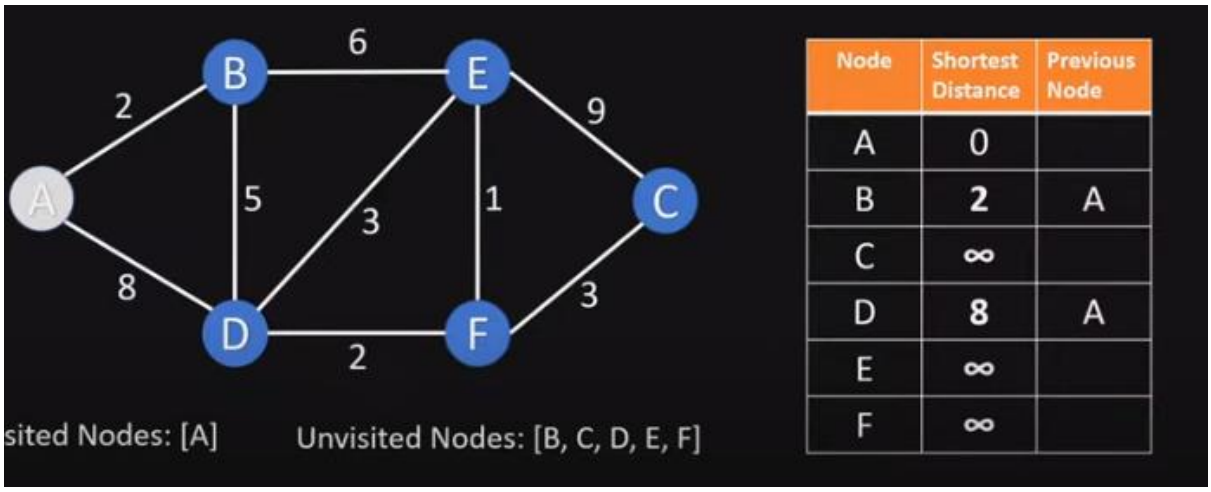
(D)

Solution: Keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted in hash table as:
For key 12, h(12) is 12%10 = 2. Therefore, 12 is placed at 2nd index in the hash table.
For key 18, h(18) is 18%10 = 8. Therefore, 18 is placed at 8th index in the hash table.
For key 13, h(13) is 13%10 = 3. Therefore, 13 is placed at 3rd index in the hash table.
For key 2, h(2) is 2%10 = 2. However, index 2 is already occupied with 12. Therefore, using linear probing, 2 will be placed at index 4 as index 2 and 3 are already occupied.
For key 3, h(3) is 3%10 = 3. However, index 3 is already occupied with 13. Therefore, using linear probing, 3 will be placed at index 5 as index 3 and 4 are already occupied.
Similarly, 23, 5 and 15 will be placed at index 6, 7, 9 respectively.

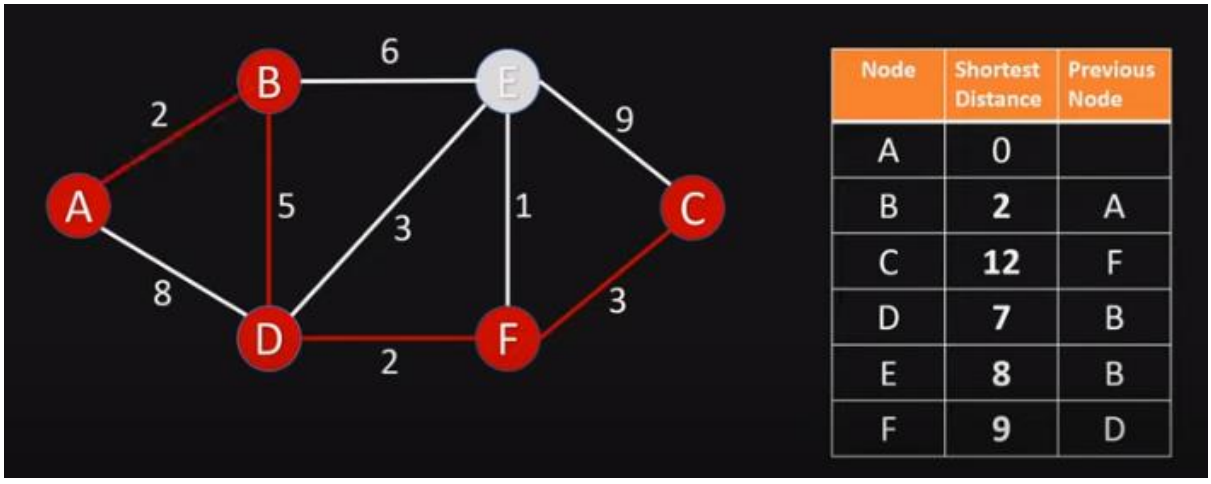
14.a. Shortest Path using Dijitras path



Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

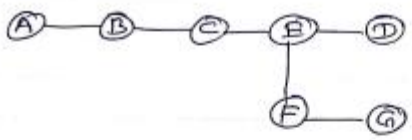


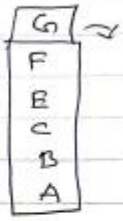
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

A-B-D-F-C = 13

14.b. BFS and DFS

DFS → Stack





BFS → queue