**NAME: ADWAIT S PURAO**

**UID: 2021300101**

**EXP NO. : 9**

**AIM: To perform Heapify(Top Down approach) , Insertion and Heapsort functions on a Heap.**

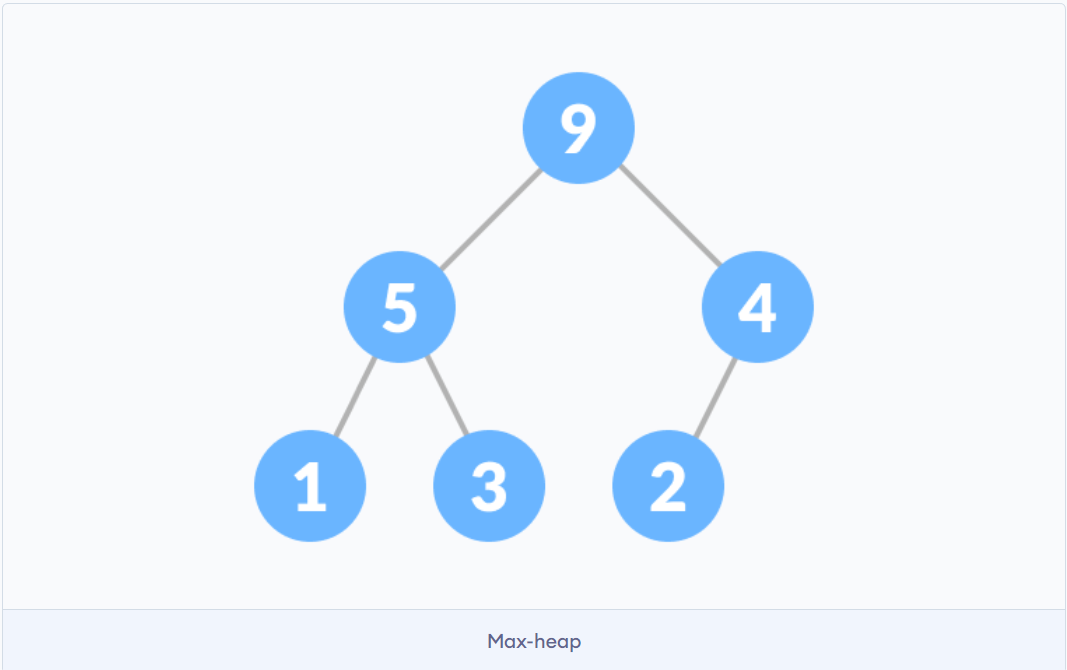
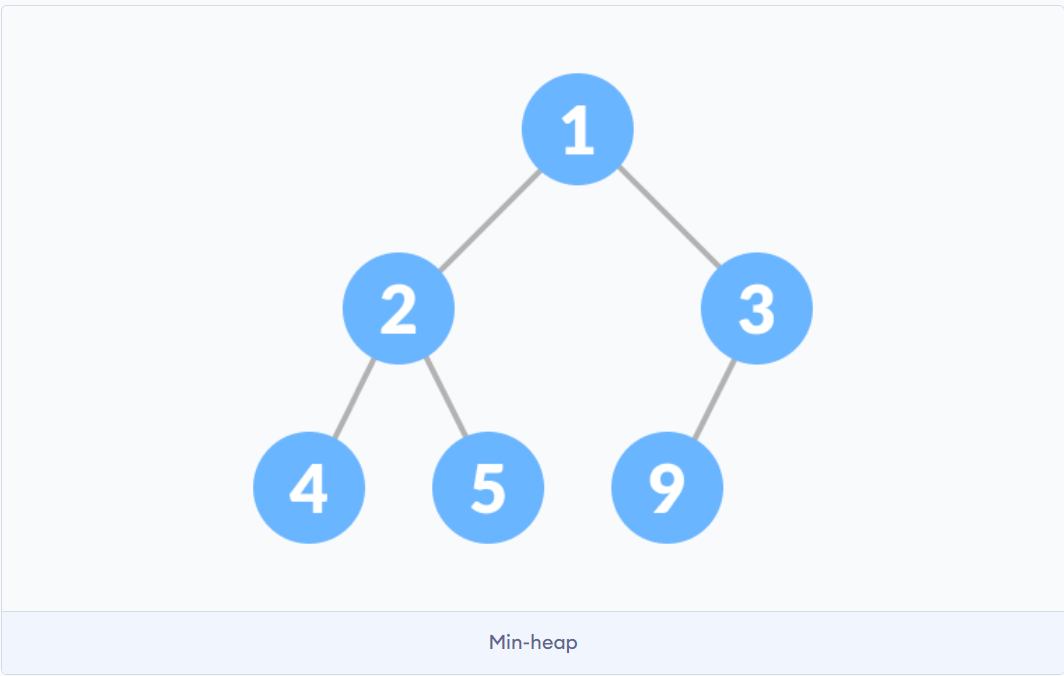
**THEORY:**

**Heap Data Structure**

**Heap data structure is a complete binary tree that satisfies the heap property, where any given node is**

**always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.**

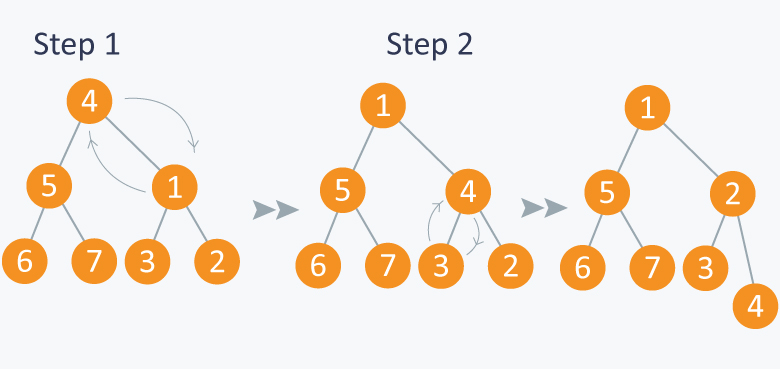
**always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.**

** **

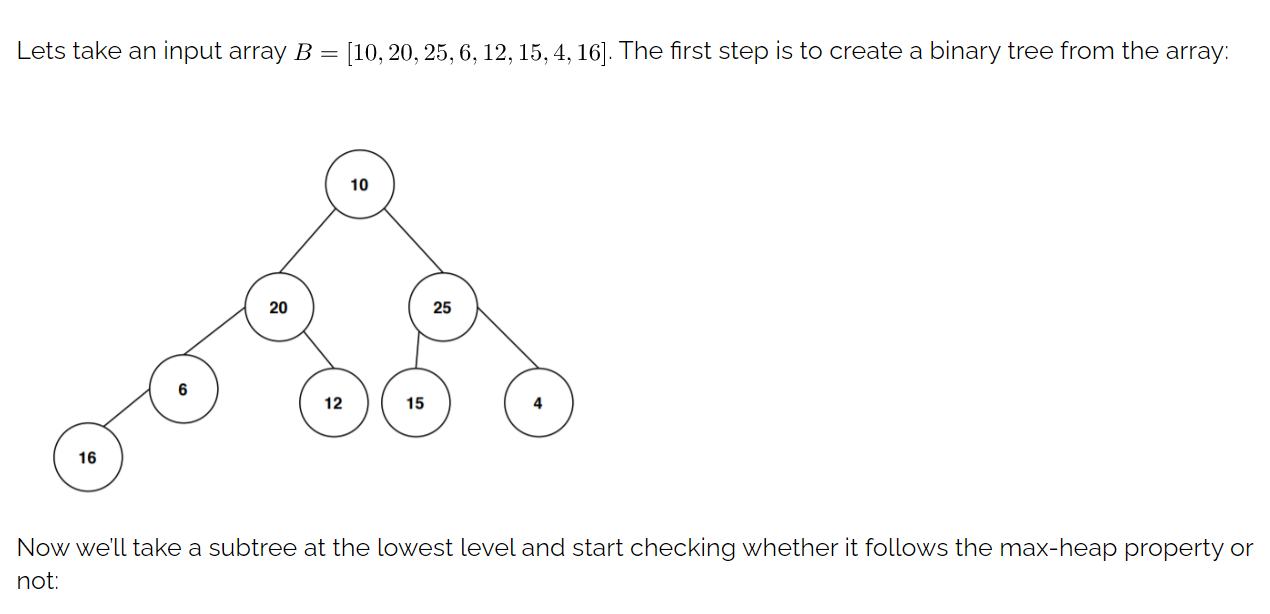
**Min Heap:**

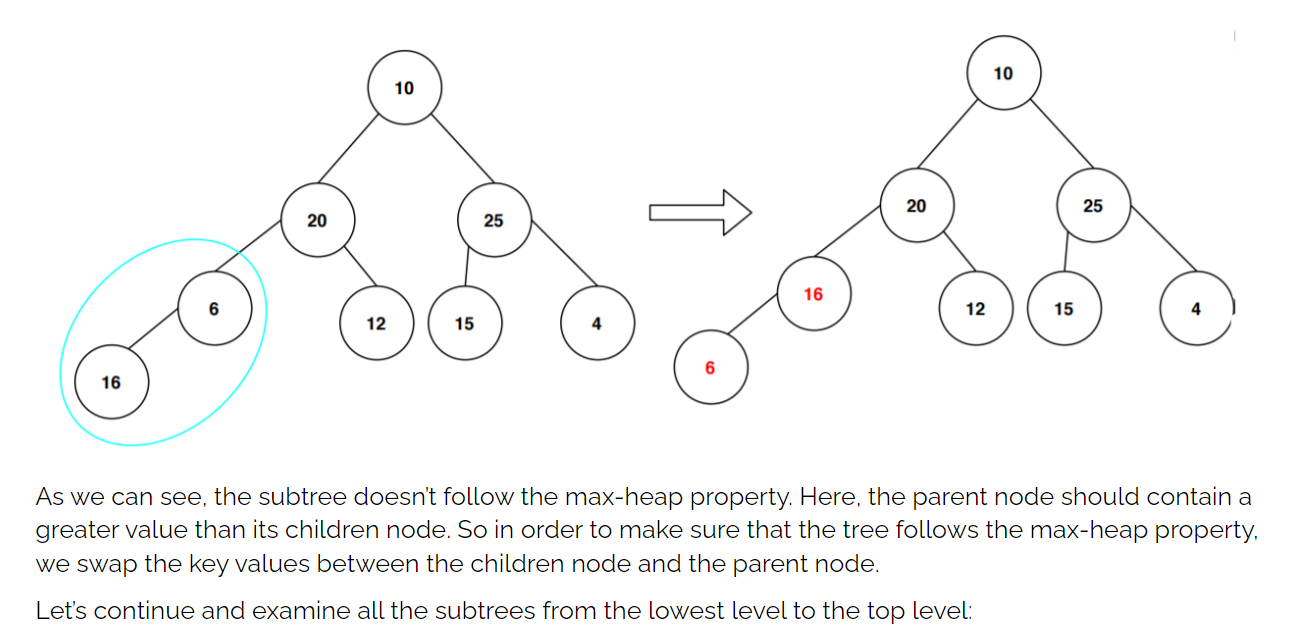
**In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.**

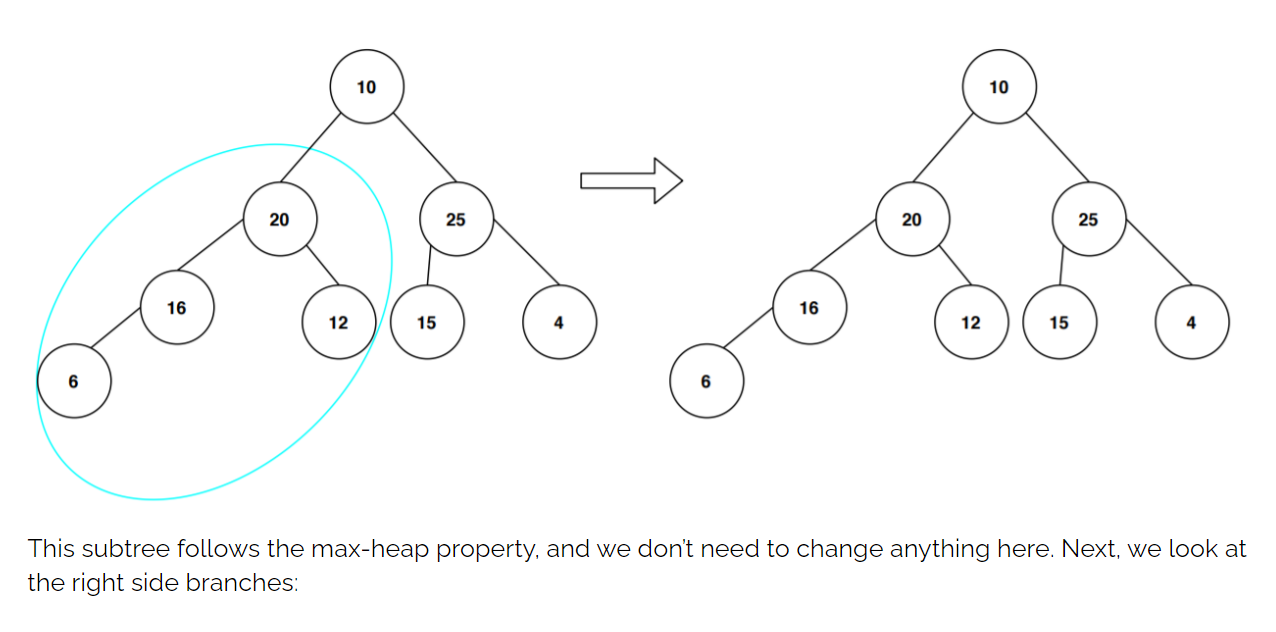
**Suppose you have elements stored in array {4, 5, 1, 6, 7, 3, 2}. As you can see in the diagram below, the element at index 1 is violating the property of min -heap, so performing min\_heapify(Arr, 1) will maintain the min-heap.**

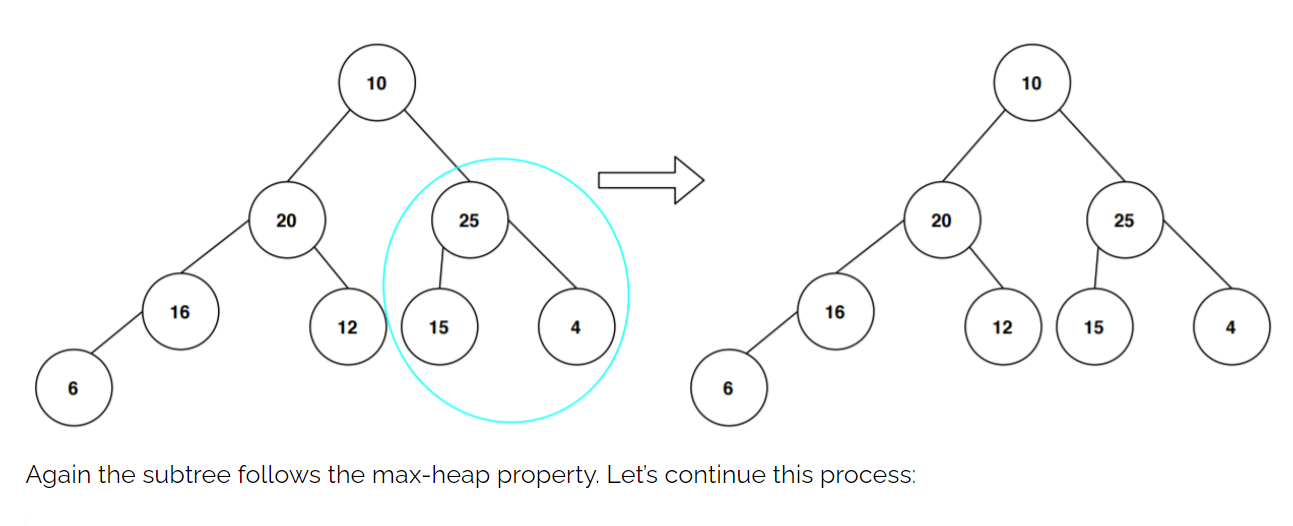


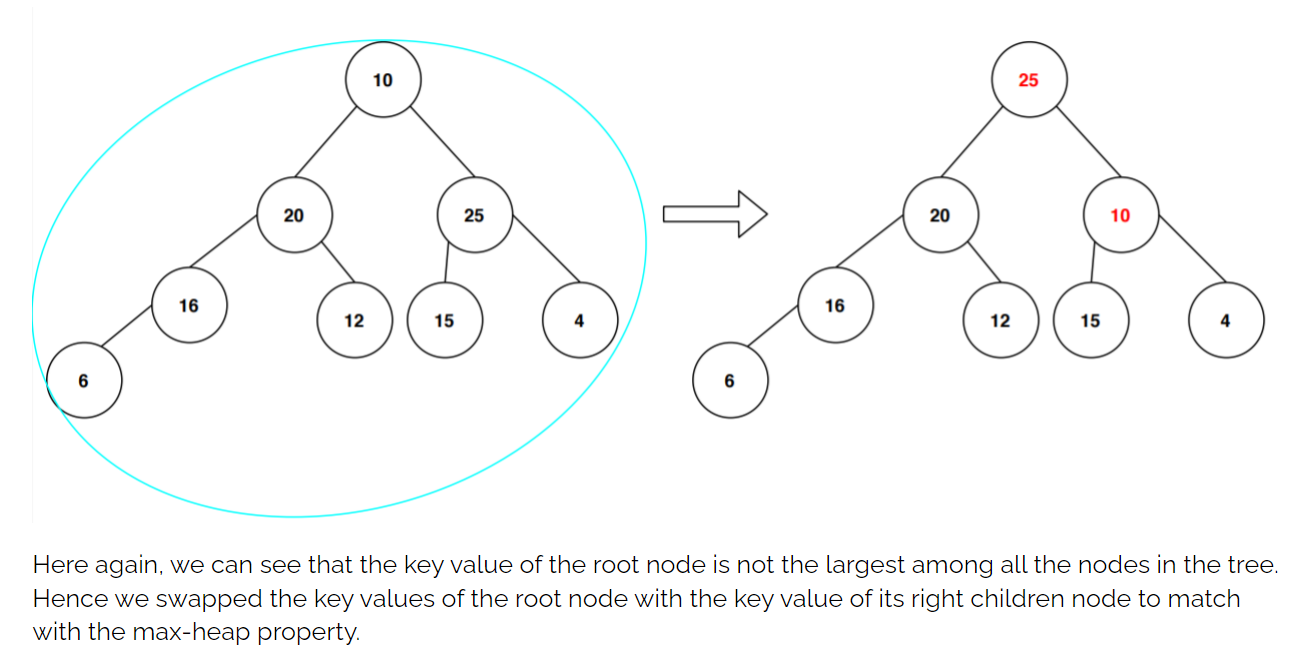
**Max-Heapify**

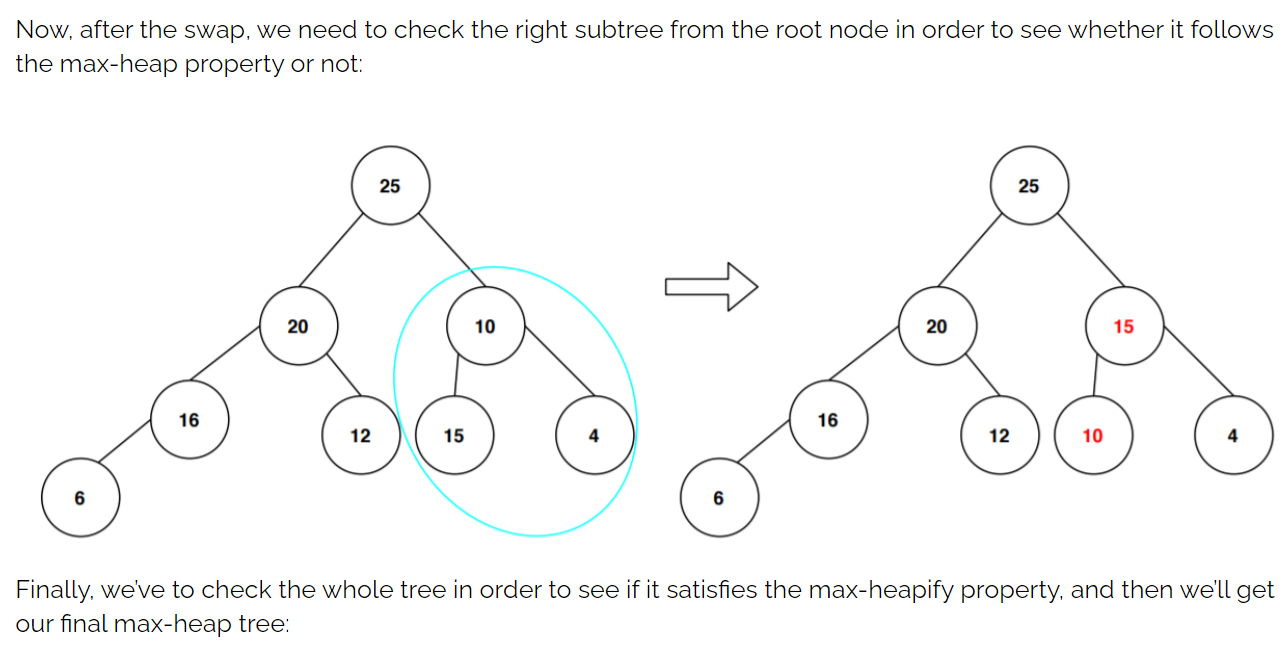
****

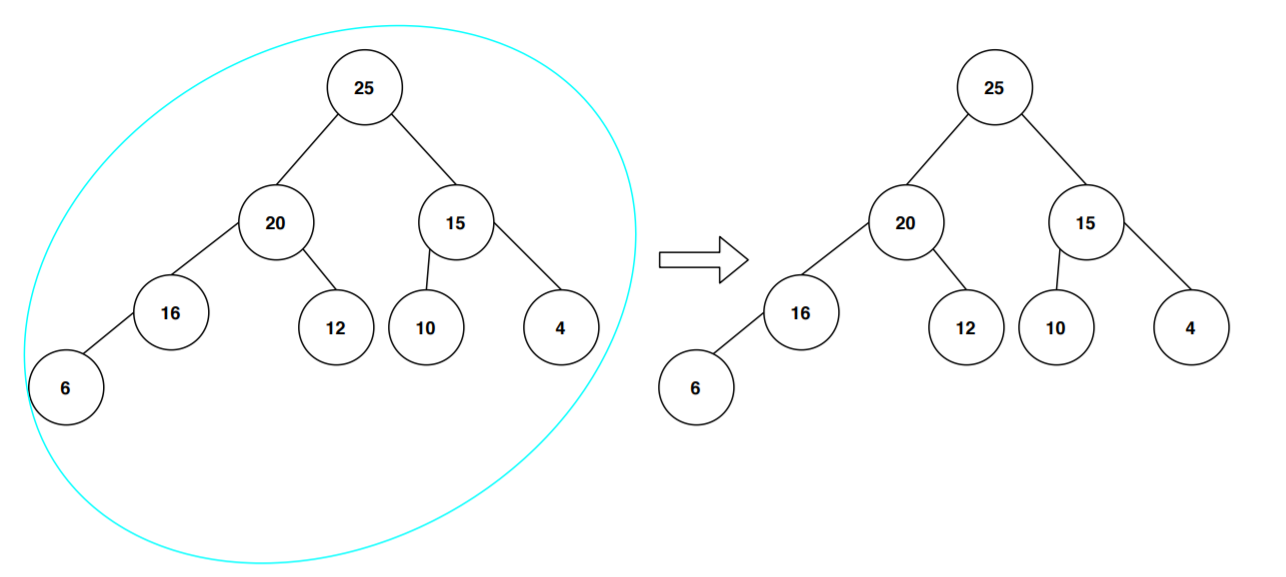
****

****

****

****

****

****

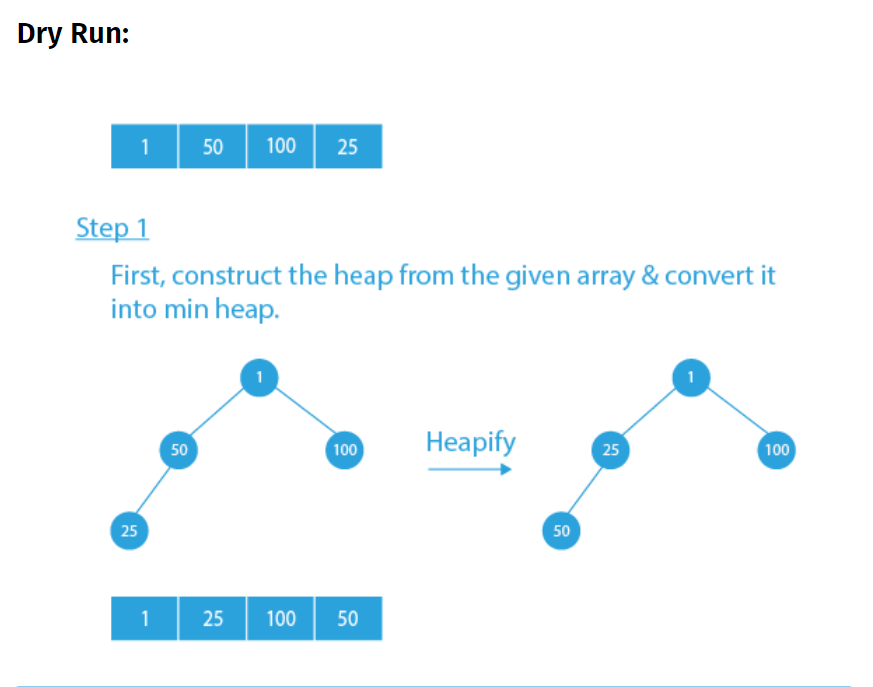
**Algorithm:**

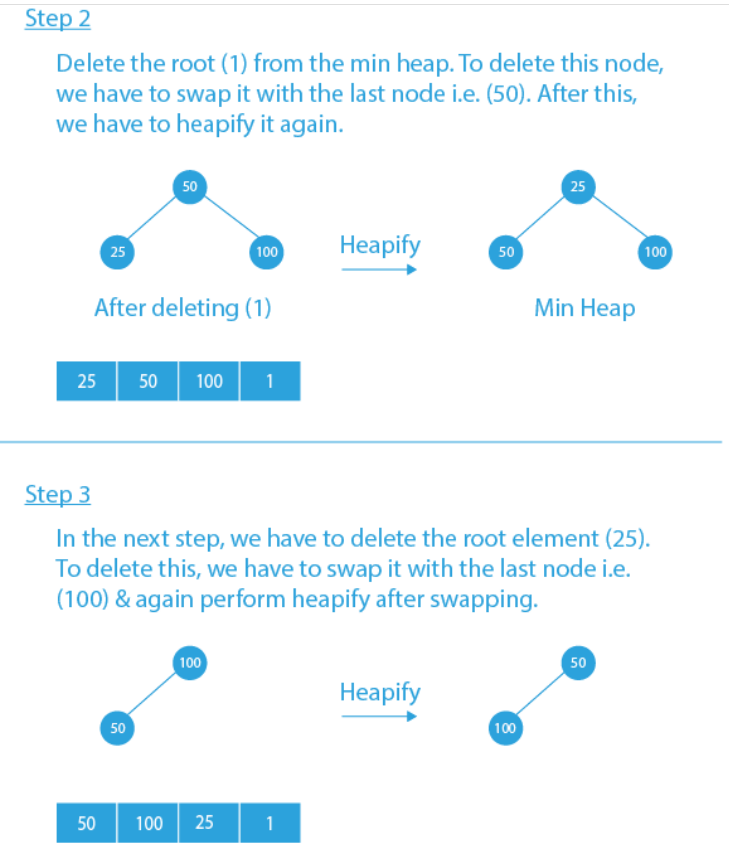
**Build a min-heap from the given input array.**

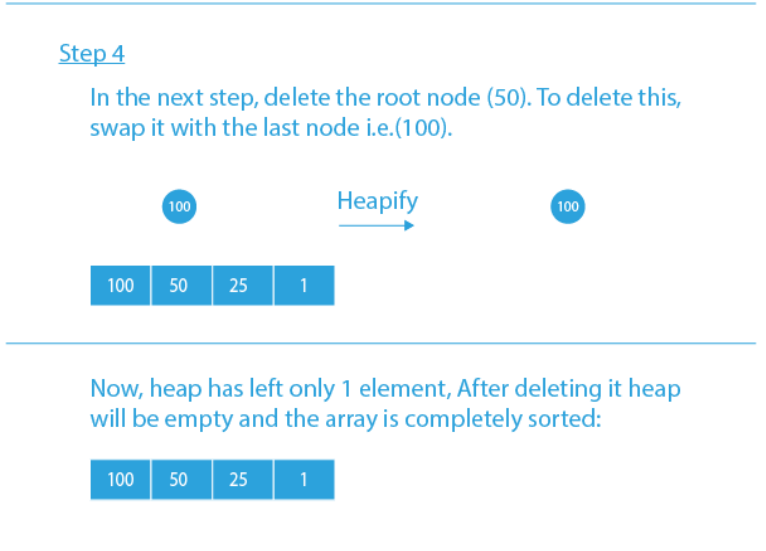
**After this, the smallest node is stored at the root of the heap. Replace it with the last node of the heap until the size of the heap gets 1.**

**Heapify the root of the tree.**

**Repeat the above steps while the size of the heap is greater than 1.**

****

****

****

**ALGORITHM:**

* **Struct Heap to maintain the heap**

**Declare a pointer to an array**

**Size variable to store the total size of array**

**Usize variable to store the Used size of array**

* **Function Swap pass integer pointers a and b**

**Store a in a temporary variable**

**Store the b in a**

**Store temp in b**

* **Function int isFull pass Struct Heap**

**If usize == size -1**

**Return 1**

**Return 0**

* **Function void Display pass struct Heap**

**Iterate a loop from i=1 until i<=usize**

**Print arr[i]**

* **Function void TDHeapify pass struct Heap**

**If usize==1**

**Return**

**Else**

**Iterate a loop from i=2 until i=usize**

**Store I in curr\_ind**

**While arr[curr\_ind]<arr[curr\_ind/2] and curr\_ind>1**

**Swap arr[curr\_ind] and arr[curr\_ind/2]**

**Make curr\_ind=curr\_ind/2**

* **Function Insertion pass struct Heap and data to be inserted**

**If heap array isFull**

**Print Heap is full**

**Else**

**Increment usize**

**Store data in arr[usize]**

**Heapify the array**

**Display the array**

* **Function int HeapSort pass struct Heap and a arrayvsorted to store the sorted elements**

**Repeat until usize!=1**

**Set count to 0**

**Sorted[count]🡨arr[1]**

**Swap arr[1] and arr[usize]**

**Decrement usize**

**Heapify the tree**

**Display the heap**

**Increment count**

**Sorted[count]🡨arr[1]**

**Return count**

* **Main function**

**Allocate memory for struct Heap**

**Take input n size of heap array**

**Set size to n+1**

**Set usize to 0**

**Allocate memory for heap array**

**Initialize all elements of heap array to 0**

**Declare an array sorted to store the sorted elements**

**Repeat until flag ==0**

**Take the choice of user**

**If choice =1**

**Take input the element to be inserted**

**Insert in heap**

**Else if choice =2**

**HeapSort the heap array**

**Display the sorted array**

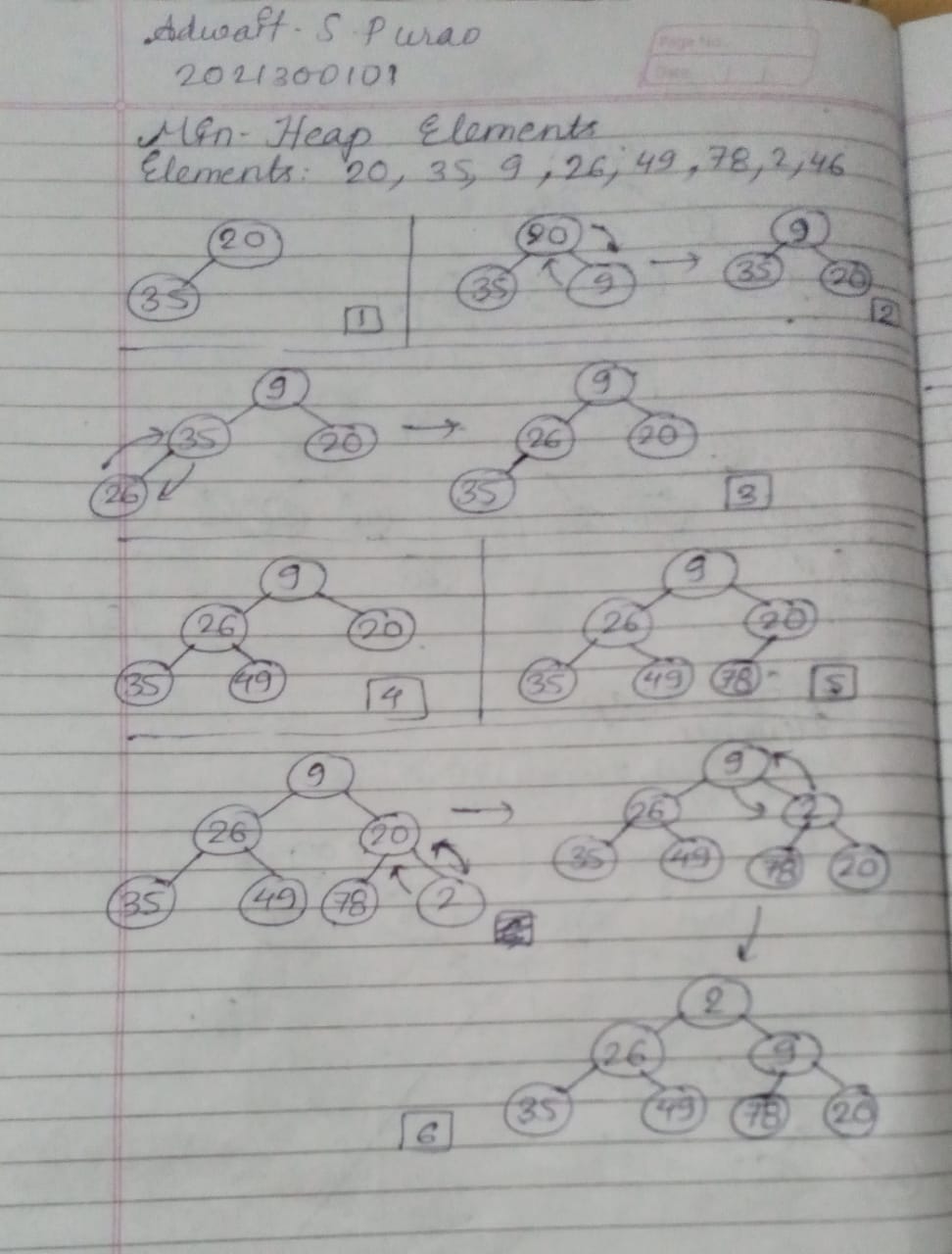
**Else if choice=3**

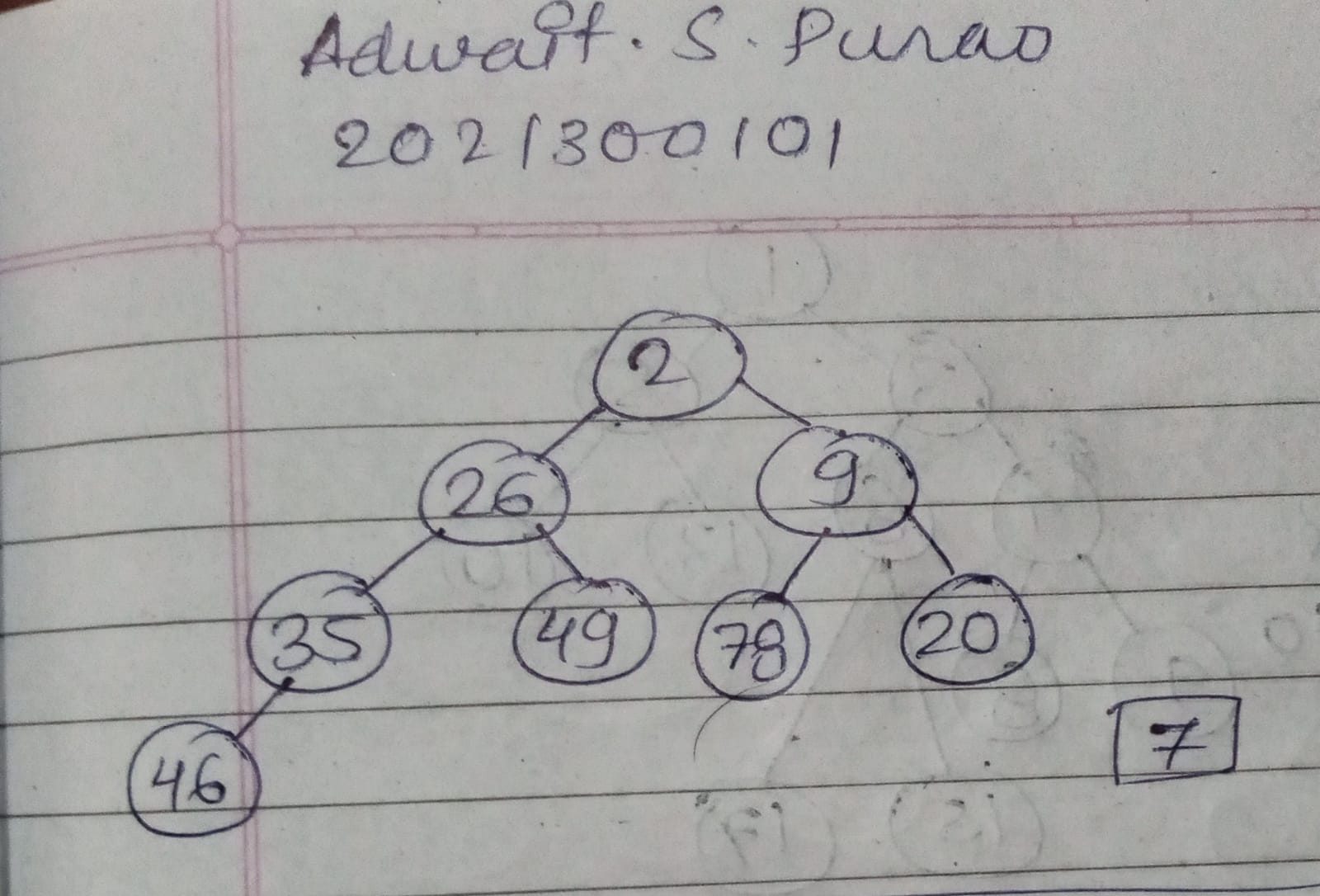
**Set flag=1**

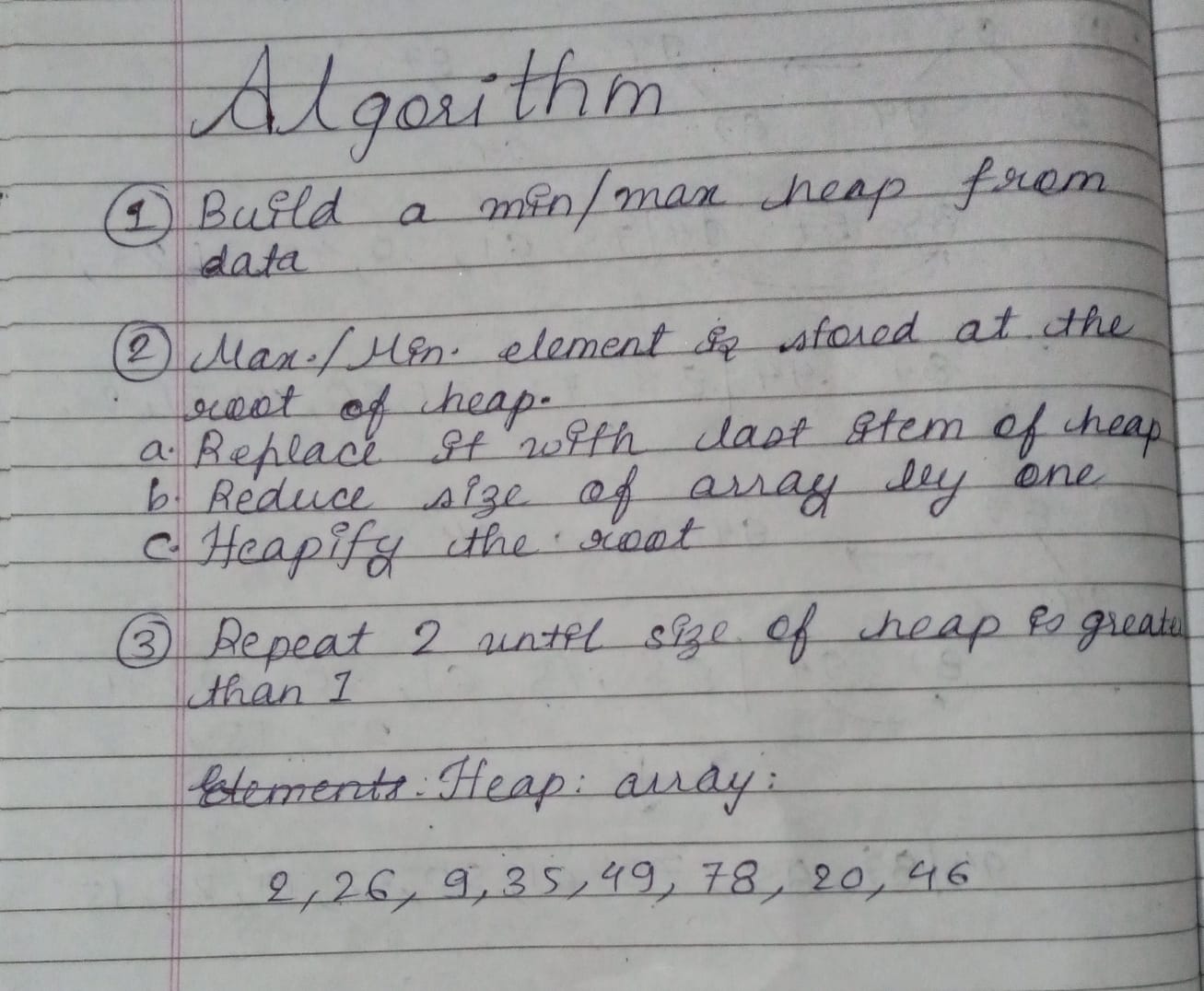
**Else**

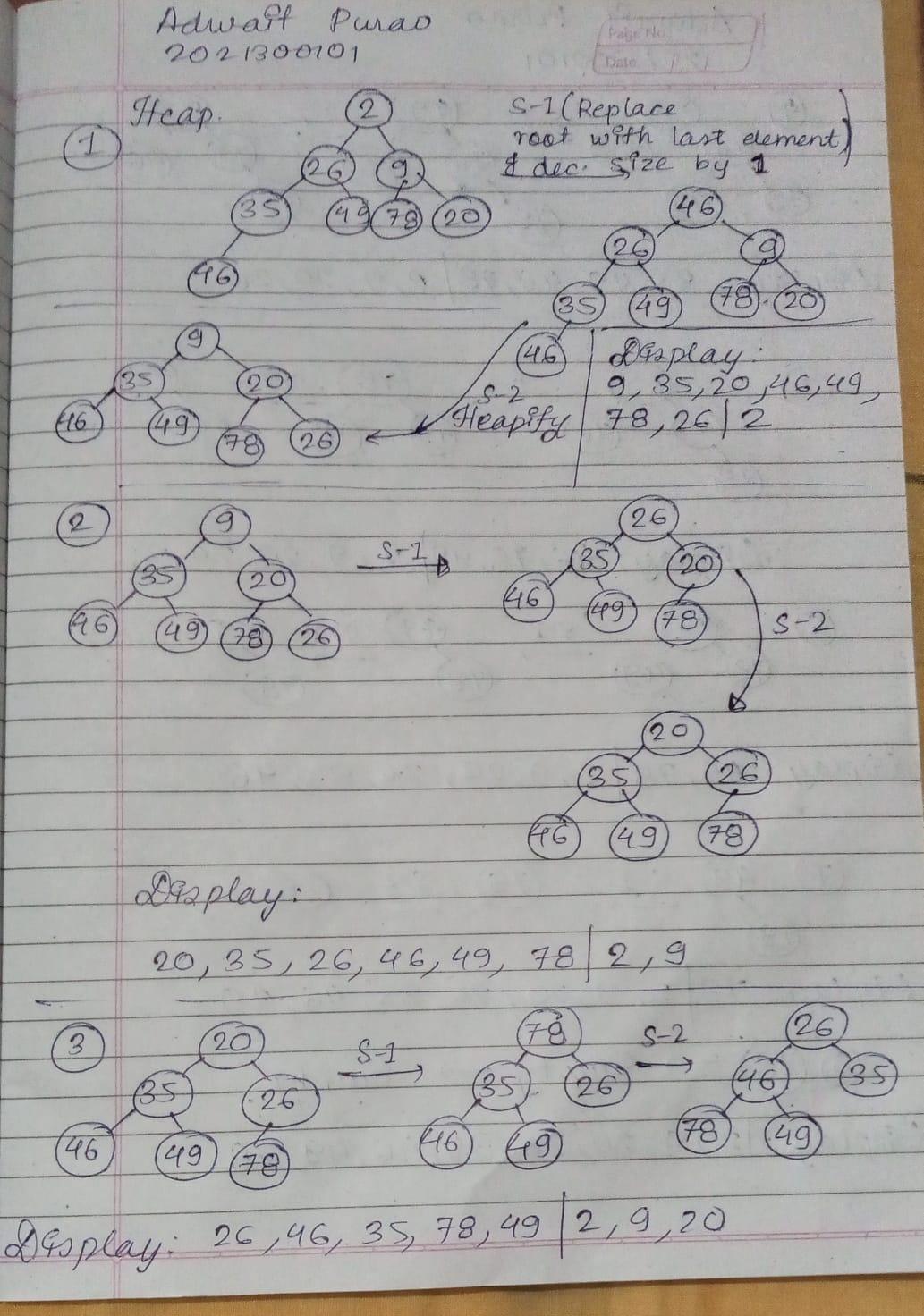
**Print Invalid input**

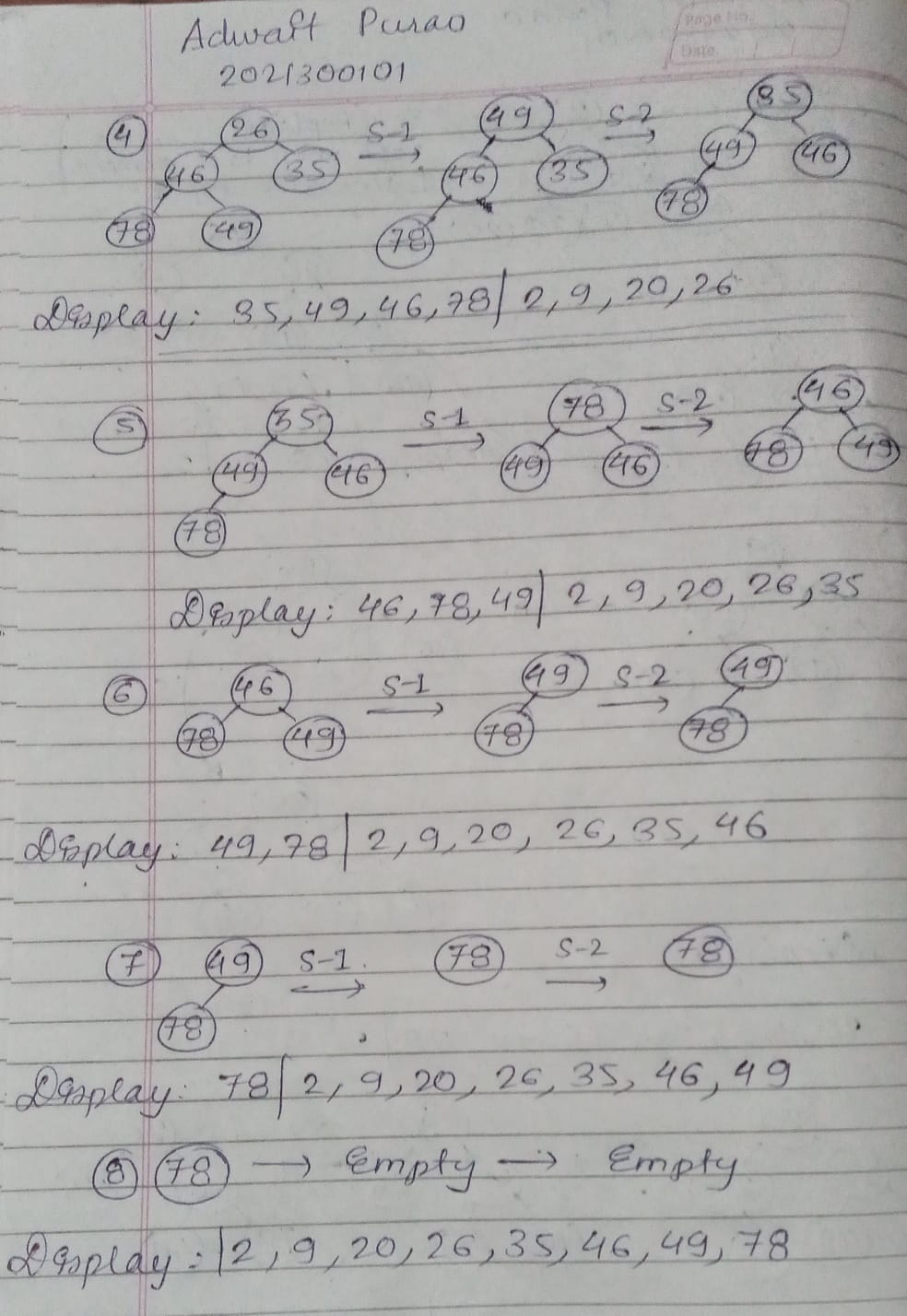
**PROBLEM SOLVING ON CONCEPT:**

****

****

****

****

****

**CODE:**

#include <stdio.h>

#include <stdlib.h>

struct Heap

{

    int \*arr;//Array to maintain the heap

    int size;//Total size of heap array

    int usize;//Used(Current) size of heap array

};

void swap(int \*a, int \*b)

{

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int isFull(struct Heap \*h)

{

    if (h->usize == h->size - 1)

    {

        return 1;

    }

    return 0;

}

void Display(struct Heap \*h)

{

    printf("\nDisplay of the Heap:\n");

    for (int i = 1; i <= h->usize; i++)

    {

        printf("%d ", (h->arr[i]));

    }

}

//Heapify Top-Down approach

void TDHeapify(struct Heap \*h)

{

    //If size of heap is already 1 ,There is no need to heapify the tree as it is already heapified

    if (h->usize == 1)

    {

        return;

    }

    else

    {

        for (int i = 2; i <= h->usize; i++)

        {

            int curr\_ind = i;

            // If parent is greater than current index max heap property gets violated , so we need to swap them

            while ((curr\_ind > 1) && (h->arr[curr\_ind] < h->arr[curr\_ind / 2]))

            {

                swap(&(h->arr[curr\_ind]), &(h->arr[curr\_ind / 2]));

                curr\_ind = curr\_ind / 2;

            }

        }

    }

}

void Insertion(struct Heap \*h, int data)

{

    if (isFull(h) == 1)

    {

        printf("Heap is full!,Can't insert anymore elements\n");

    }

    else

    {

        h->usize++;

        h->arr[h->usize] = data;

        TDHeapify(h);

    }

    Display(h);

}

int HeapSort(struct Heap \*h, int sorted[])

{

    int count = 0;

    while (h->usize != 1)

    {

        count++;

        sorted[count] = h->arr[1];

        swap(&(h->arr[h->usize]), &(h->arr[1]));//Replace last element of heap with the first

        h->usize--;

        TDHeapify(h);//Heapify the tree

        Display(h);

    }

    count++;

    sorted[count] = h->arr[1];

    return count;

}

int main()

{

    struct Heap \*h = (struct Heap \*)malloc(sizeof(struct Heap));

    int n;

    int element;

    printf("Enter the size of Heap array\n");

    scanf("%d", &n);

    h->size = n + 1;

    h->usize = 0;

    h->arr = (int \*)malloc(h->size \* sizeof(int));

    for (int i = 1; i < h->size; i++)

    {

        h->arr[i] = 0;

    }

    int sorted[n + 1];//Array to store the sorted elements

    int flag = 0;

    do

    {

        printf("Enter your choice:\n1)Insertion\n2)HeapSort\n3)Exit\n");

        int ch;

        scanf("%d", &ch);

        if (ch == 1)

        {

            printf("Enter the element to be inserted:\n");

            scanf("%d", &element);

            Insertion(h, element);

            printf("\n");

        }

        else if (ch == 2)

        {

            int count = HeapSort(h, sorted);

            printf("\nThe elements in sorted order\n");

            for (int i = 1; i <= count; i++)

            {

                printf("%d ", sorted[i]);

            }

            printf("\n");

        }

        else if (ch == 3)

        {

            printf("Program finished!\n");

            flag = 1;

        }

        else

        {

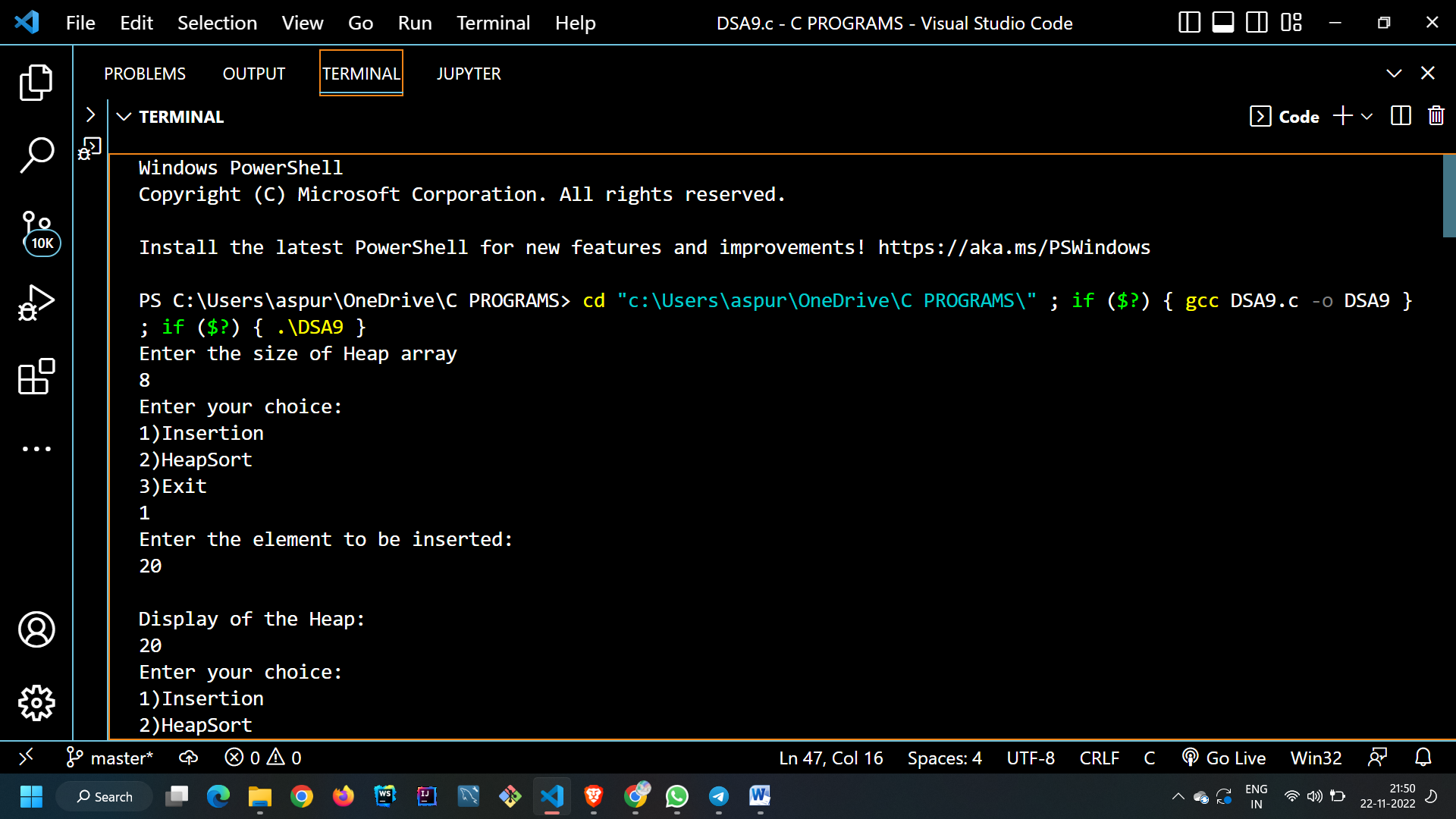
            printf("Invalid choice!\n");

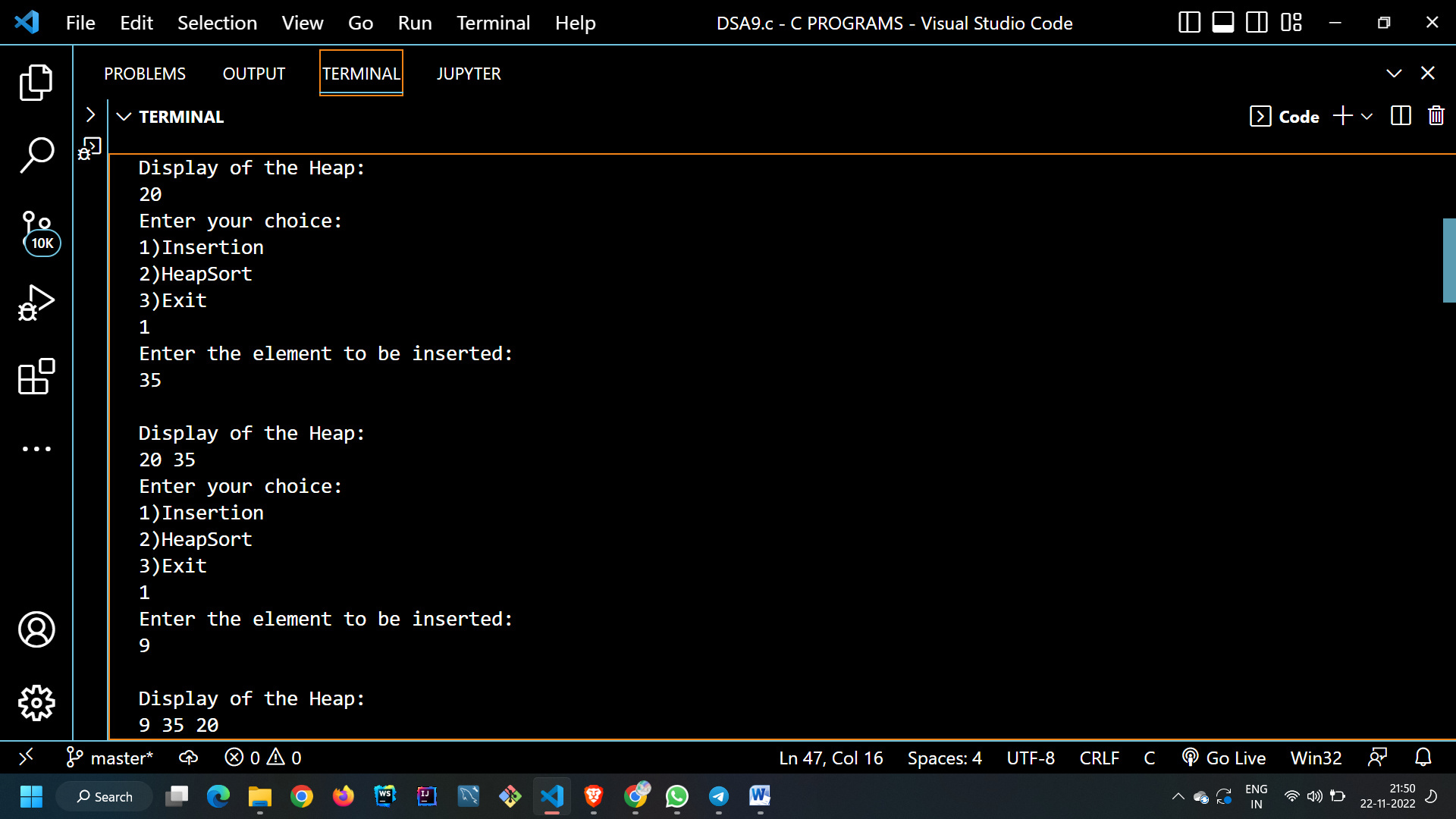
        }

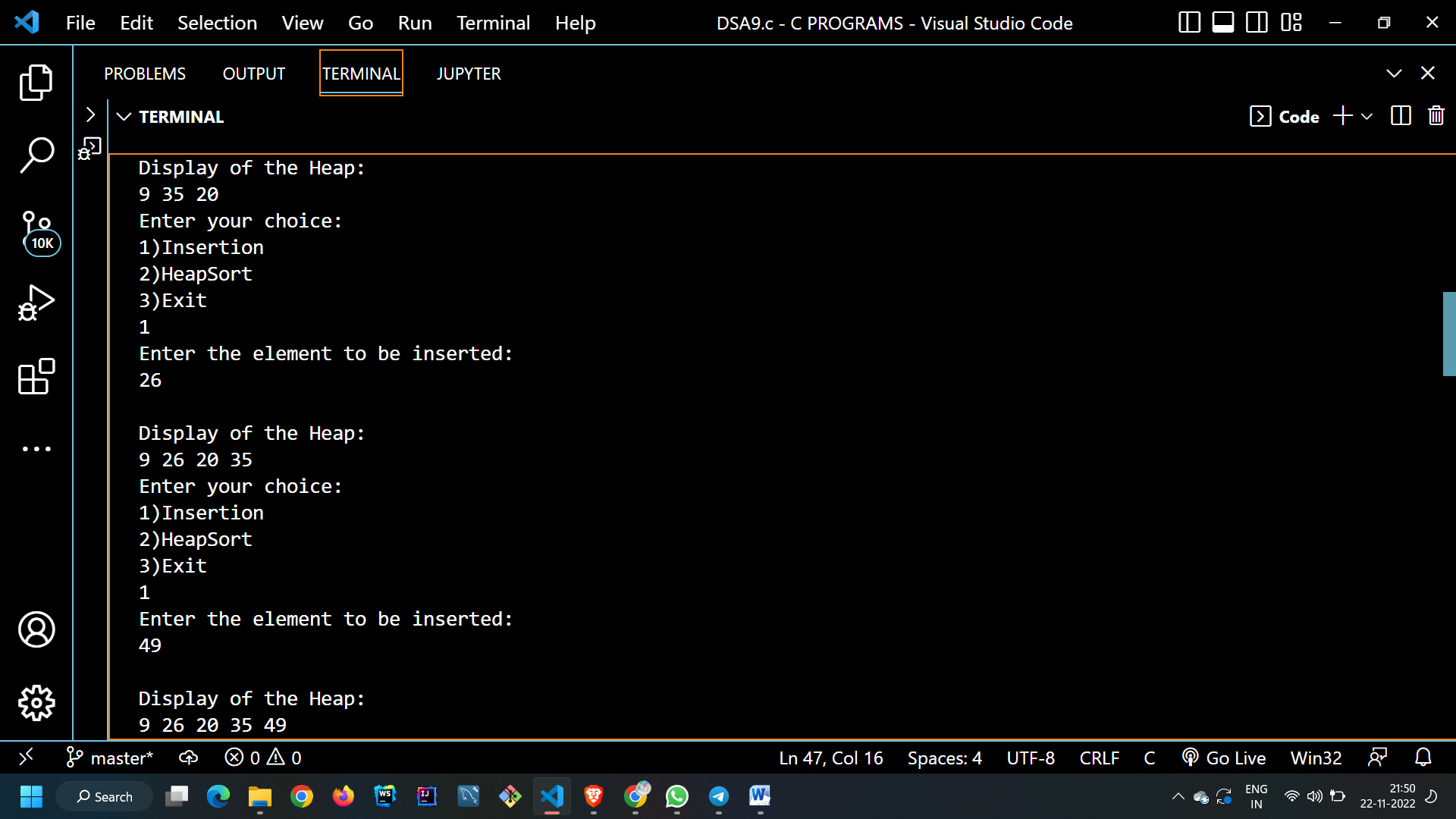
    } while (flag != 1);

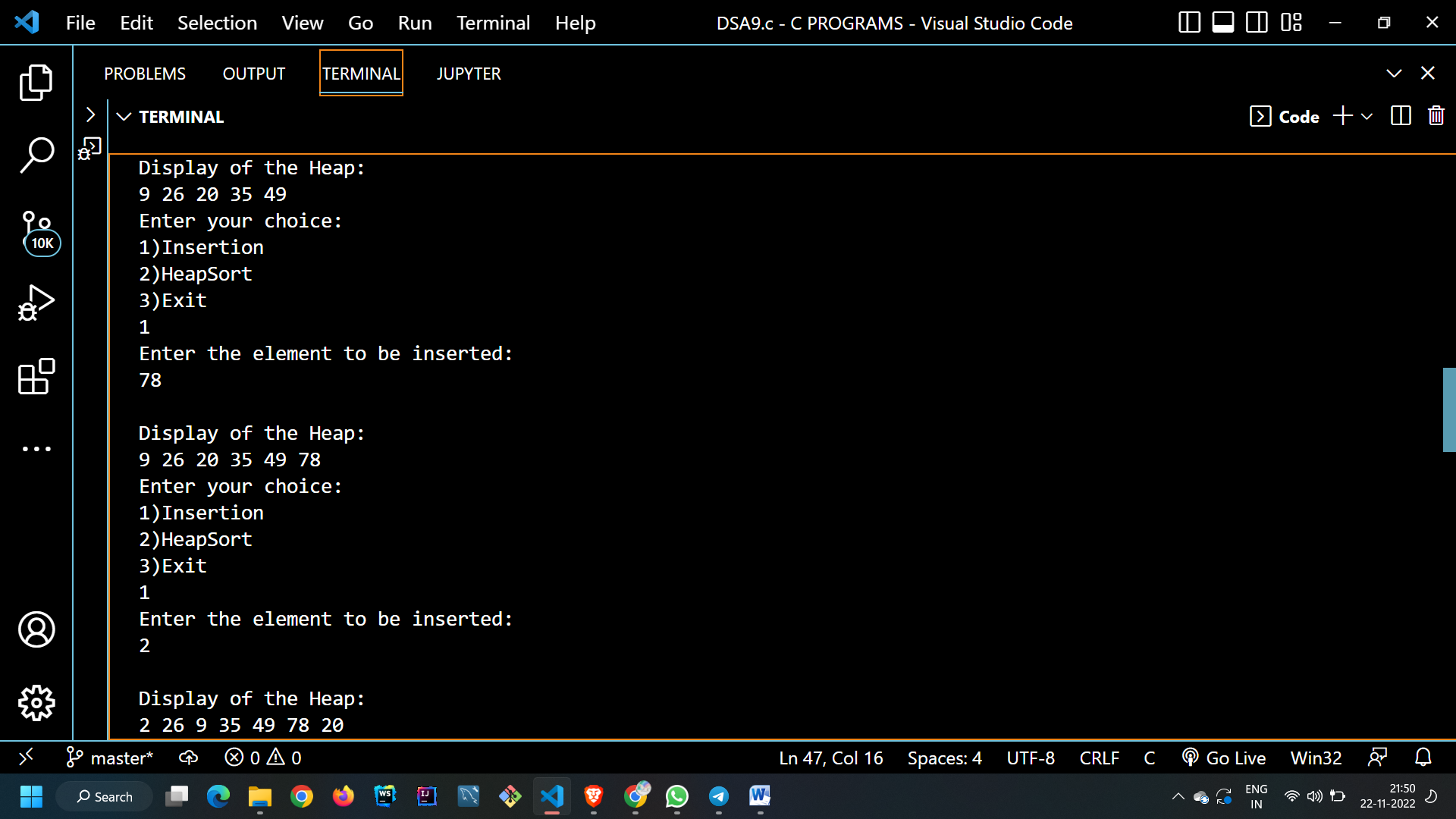
}

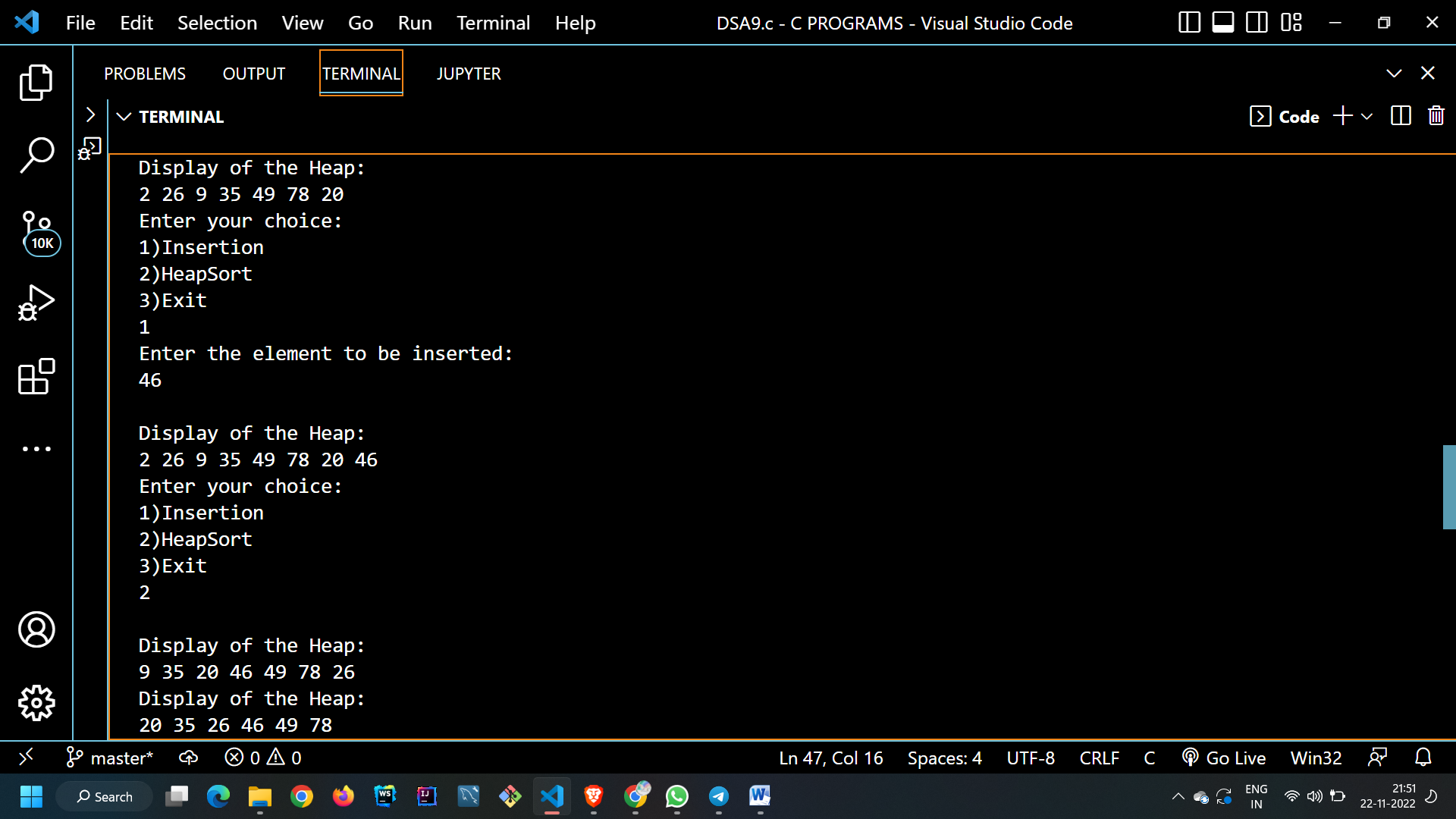
**OUTPUT SCREENSHOT:**

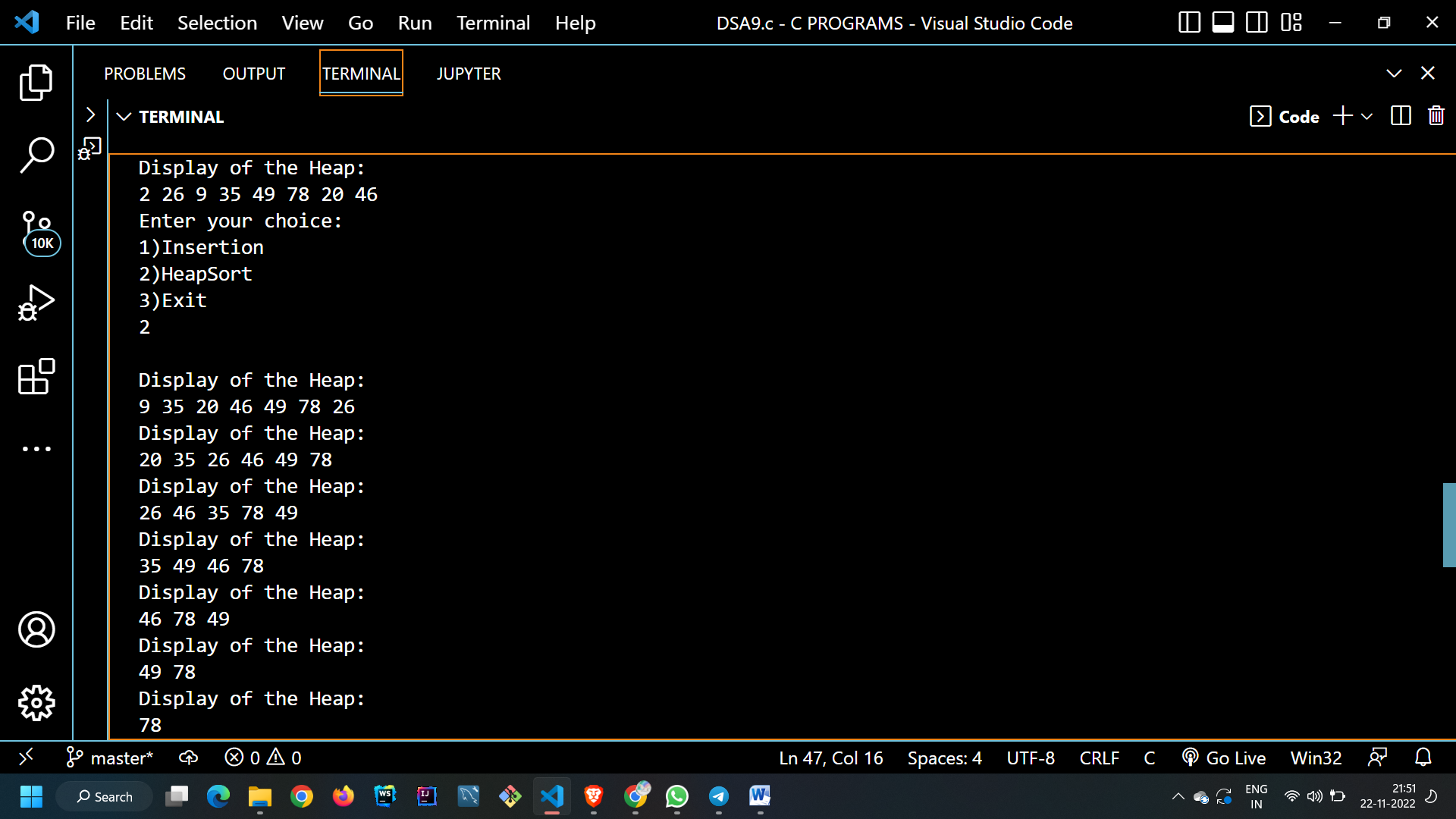


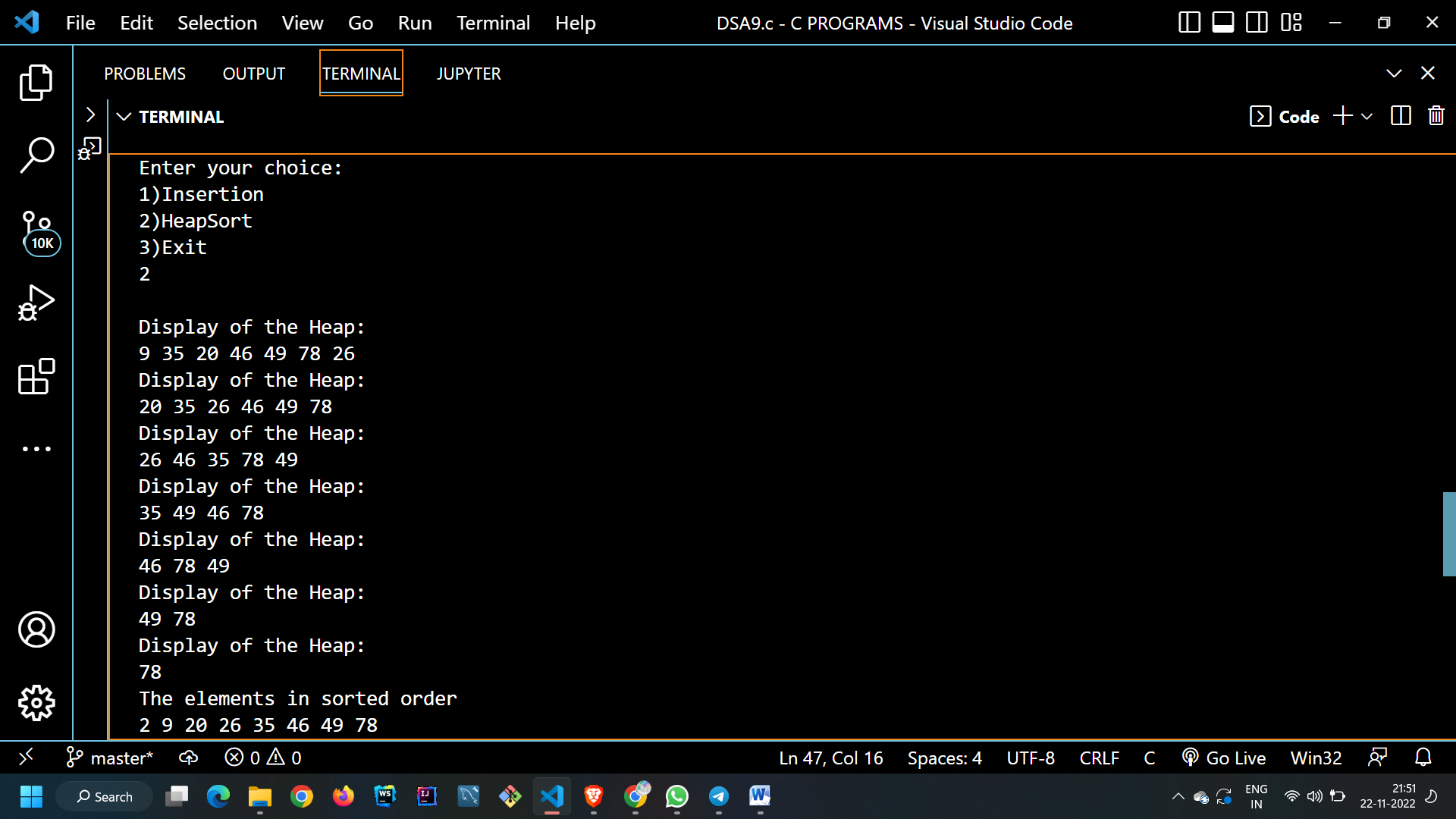




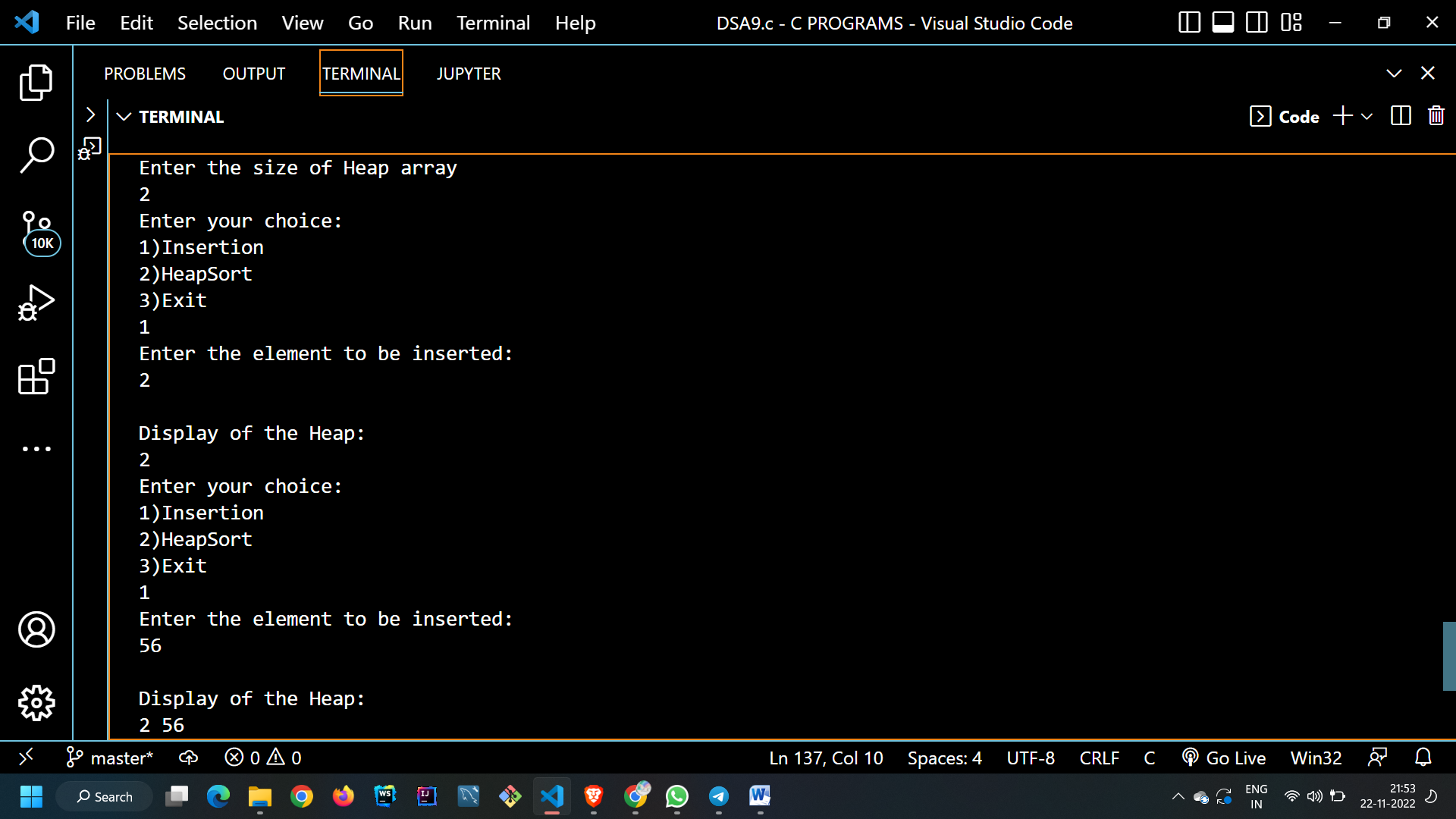


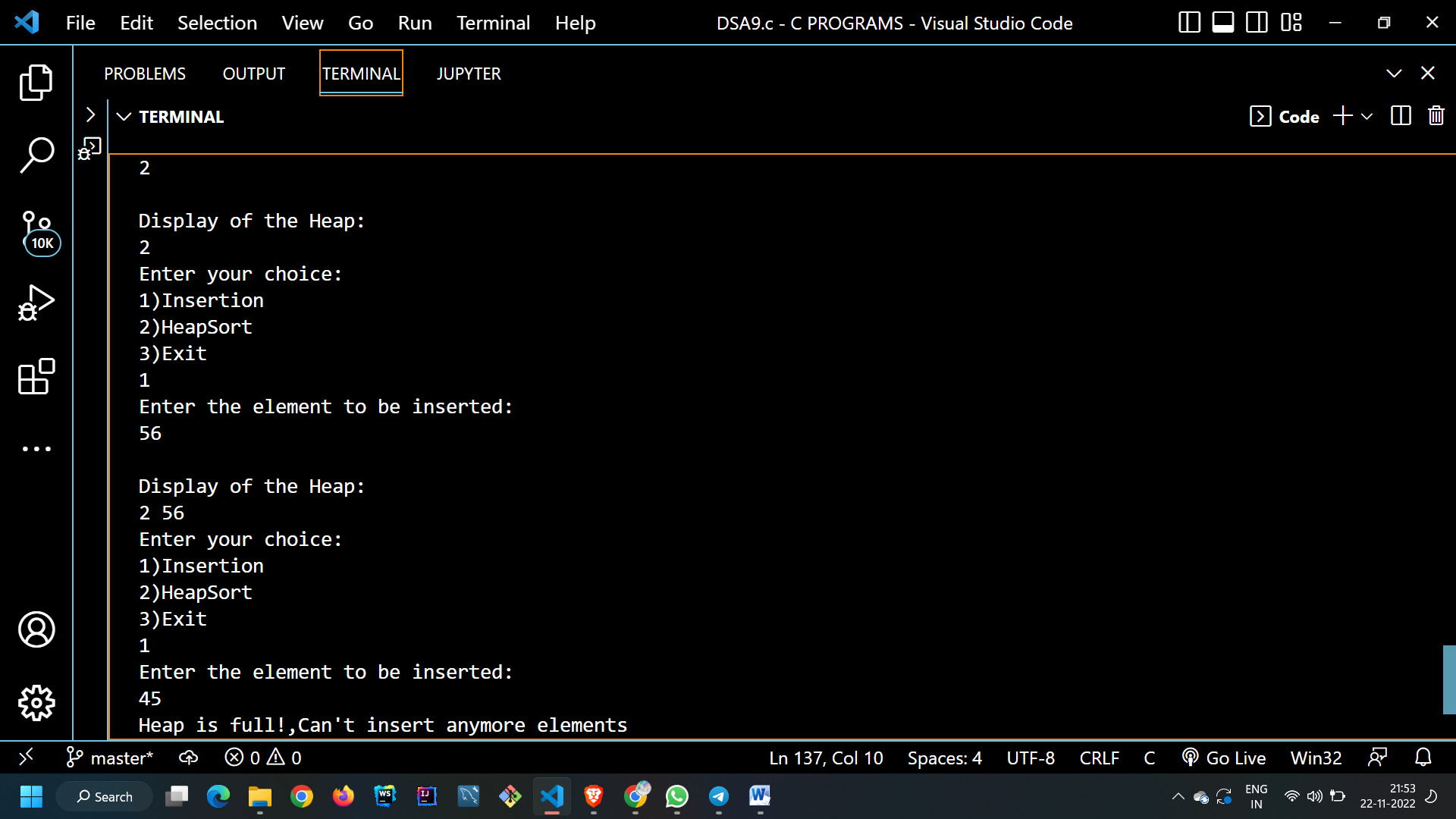




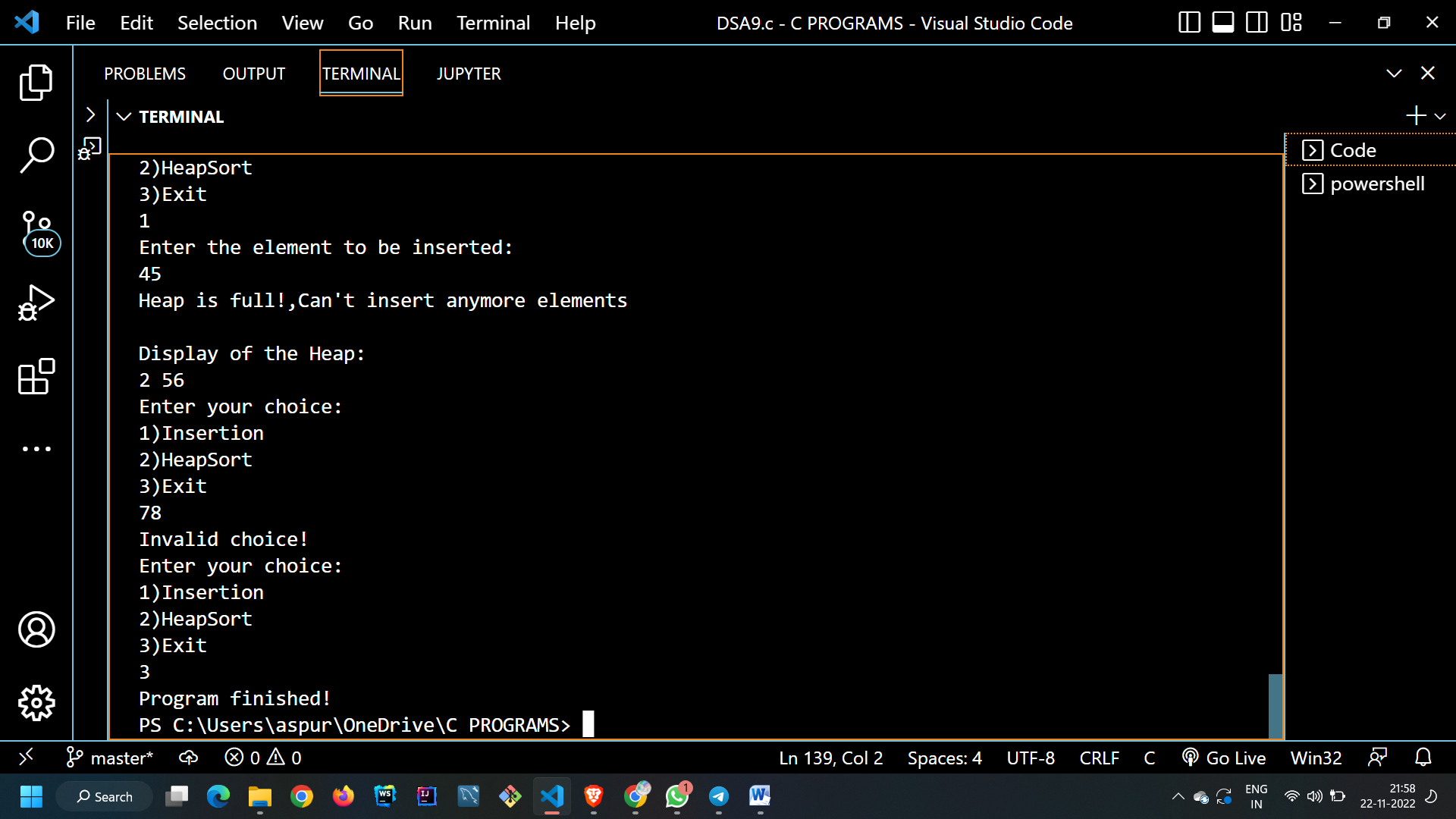


**Test Case : When Heap is Full**





**Test Case: Invalid Choice**



**CONCLUSION:**

In this experiment we learnt about Heap data structure and it’s application . We learnt the implementation of Heaps using arrays . We learnt the internal structure of a Heap structure.

In the end we implemented Insertion , Heapify(Top Down Approach) , Heapsort functions with the help of a menu driven program in C.