HEAP:-

Problem Statement

Michael is developing a navigation system for a delivery robot where new delivery locations are frequently added to the system. He requires a function to assist him in inserting these new delivery locations into a binary min-heap, based on their distances from the current location.

Write a code to assist Michael in inserting new delivery locations (represented as integers indicating distances) into a binary min-heap.

Input format :

The input consists of a series of space-separated integers, representing the distance of locations, that are to be inserted into the heap.

Output format :

The output prints the binary min-heap, after inserting all the distances of locations.

Code constraints :

The maximum heap size is defined by maxSize (here set to 100).

Sample test cases :

Input 1 :

```
5 10 3 1 7
```

Output 1 :

```
1 3 5 10 7
```

Input 2 :

```
15 8 12 6 4 9
```

Output 2 :

```
4 6 9 15 8 12
```

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

struct Location {
    int distance;
};

void swap(Location& a, Location& b) {
    Location temp = a;
```

```
    a = b;
    b = temp;
}

// Function to insert a new location into the binary heap
void insertLocation(Location* heap, int& heapSize, Location newLocation) {
    heapSize++;

    int i = heapSize - 1;
    heap[i] = newLocation;

    while (i > 0 && heap[i].distance < heap[(i - 1) / 2].distance) {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printHeap(const Location* heap, int heapSize) {
    for (int i = 0; i < heapSize; i++) {
        cout << heap[i].distance << " ";
    }
    cout << endl;
}
```

```
int main() {
    const int maxHeapSize = 100;
    Location* binaryHeap = new Location[maxHeapSize];
    int heapSize = 0;

    while (true) {
        Location newLocation;
        if (!(cin >> newLocation.distance)) {
            break;
        }

        insertLocation(binaryHeap, heapSize, newLocation);
    }

    printHeap(binaryHeap, heapSize);

    delete[] binaryHeap;

    return 0;
}
```

Problem Statement

Ethan is working on an employee assignment management system for a company. Employees are assigned tasks based on their expertise and workload. He needs to implement a program that allows for the assignment of tasks to employees while ensuring that the employees with the highest expertise are assigned tasks first.

Write the function to help him insert a new task (an integer representing the complexity of task) into a max heap.

Input format :

The input consists of a series of space-separated integers, representing the complexity of tasks, and inserts them into the max heap.

Output format :

The output prints the max heap after each task insertion, showing the current state of the heap.

Code constraints :

The maximum heap size is 100 (defined by maxHeapSize).

Sample test cases :

Input 1 :

```
8 3 6 10 5 2
```

Output 1 :

```
10 8 6 3 5 2
```

Input 2 :

```
50 30 70 20 40
```

Output 2 :

```
70 40 50 20 30
```

```
#include <iostream>

using namespace std;


void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}
```

```
// Function to insert a new task into the max heap
void insertTask(int heap[], int& heapSize, int newTask) {
    heapSize++;

    int i = heapSize - 1;
    heap[i] = newTask;

    while (i > 0 && heap[(i - 1) / 2] < heap[i]) {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printHeap(int heap[], int heapSize) {
    for (int i = 0; i < heapSize; i++) {
        cout << heap[i] << " ";
    }
    cout << endl;
}

int main() {
    int maxHeapSize = 100;
    int maxHeap[maxHeapSize];
    int heapSize = 0;
```

```
    while (true) {
        int newTask;
        if (!(cin >> newTask)) {
            break;
        }

        if (heapSize < maxHeapSize) {
            insertTask(maxHeap, heapSize, newTask);
        } else {
            cout << "Heap is full, cannot insert more tasks." << endl;
            break;
        }
    }

    printHeap(maxHeap, heapSize);

    return 0;
}
```

Problem Statement

A magician has placed the enchanted gemstones in an array, and your task is to convert the array into a min heap, which will help you identify the gemstone with the **k**th most potent magic.

Given an array of elements and an element **k**, find the **k**th element after converting the array into a min-heap.

Note: This question was asked in Cisco recruitment.

Input format :

The first line of input is an integer value, representing the number of elements in the array.

The second line of the input consists of space-separated integer array values.

The third line of the input is an integer value, k.

Output format :

The first line of the output prints the space-separated integer values in the min-heap.

The second line of the output prints the **k**th element in the min-heap.

If k>n, print "Invalid entry".

Refer to the sample output for formatting specifications.

Code constraints :

size of array>0

k<=size of array

Sample test cases :

Input 1 :

```
5
2 4 1 5 9
3
```

Output 1 :

```
Min heap is: 1 4 2 5 9
Kth element in min heap is 2
```

Input 2 :

```
5
2 4 1 5 9
12
```

Output 2 :

```
Invalid entry
```

```
#include <iostream>

using namespace std;
```

```
void min_heap(int *a, int m, int n) {
    int j, t;
    t = a[m];
    j = 2 * m;
    while (j <= n) {
        if (j < n && a[j + 1] < a[j])
            j = j + 1;
        if (t < a[j])
            break;
        else if (t >= a[j]) {
            a[j / 2] = a[j];
            j = 2 * j;
        }
    }
    a[j / 2] = t;
}

void build_minheap(int *a, int n) {
    int k;
    for (k = n / 2; k >= 1; k--) {
        min_heap(a, k, n);
    }
}
```

```
int main() {
    int n, i;
    cin >> n;

    int a[n + 1]; // Adjusted array size to start from index 1
    for (i = 1; i <= n; i++) {
        cin >> a[i];
    }
    int element;
    cin >> element;
    if (element < 1 || element > n) {
        cout << "Invalid entry" << endl;
        return 0;
    }
    build_minheap(a, n);
    cout << "Min heap is: ";
    for (i = 1; i <= n; i++) {
        cout << a[i] << " ";
    }
    cout << endl << "Kth element in min heap is " << a[element];
    return 0;
}
```

Problem Statement

In a mysterious cave, explorers discovered a hidden chamber filled with enchanted crystals of varying sizes. They want to organize these crystals into a min-heap to harness their power.

Write a program that helps them build the min-heap and find the k^{th} smallest crystal's size, where k is a user-defined parameter.

Input format :

The first line contains an integer n, the number of crystals.

The next n lines contain the sizes of the crystals. Each line contains a single integer representing the size of a crystal.

The last line contains an integer k, indicating the position of the crystal you want to find.

Output format :

The first line prints the elements in the min-heap after construction.

The second line prints the size of the k^{th} smallest crystal.

Refer to the sample output for formatting specifications.

Code constraints :

1 <= n <= 1000

1 <= k <= n

Sample test cases :

Input 1 :

```
5
3 5 2 6 8
12
```

Output 1 :

```
Invalid entry
```

Input 2 :

```
5
3 5 2 6 8
3
```

Output 2 :

```
Min heap is: 2 5 3 6 8
The smallest crystal's size is 3
```

```
#include <iostream>

using namespace std;


void min_heap(int *a, int m, int n) {

   int j, t;
```

```
    t = a[m];
    j = 2 * m;
    while (j <= n) {
        if (j < n && a[j + 1] < a[j])
            j = j + 1;
        if (t < a[j])
            break;
        else if (t >= a[j]) {
            a[j / 2] = a[j];
            j = 2 * j;
        }
    }
    a[j / 2] = t;
}

void build_minheap(int *a, int n) {
    int k;
    for (k = n / 2; k >= 1; k--) {
        min_heap(a, k, n);
    }
}

int main() {
    int n, i;
    cin >> n;

    int a[n + 1];
    for (i = 1; i <= n; i++) {
        cin >> a[i];
```

```
    }

    int k;
    cin >> k;
    if (k < 1 || k > n) {
        cout << "Invalid entry";
        return 0;
    }

    build_minheap(a, n);
    cout << "Min heap is: ";
    for (i = 1; i <= n; i++) {
        cout << a[i] << " ";
    }
    cout << endl << "The smallest crystal's size is " << a[k];

    return 0;
}
```

Problem Statement

The group of archaeologists studying ancient artifacts has presented Mark Antony with a perplexing challenge. They've come across an enigmatic array of elements that appear to depict a min-heap tree, but they are struggling to efficiently extract the largest element from this min-heap.

Mark Antony's task is to assist them in transforming the provided min-heap array into a max-heap representation. This conversion will facilitate the straightforward retrieval of the largest element from the tree.

Note: This kind of question will be helpful in clearing Accenture recruitment.
Input format :

The first line of input consists of an integer **n**, representing the number of elements in the min heap tree.
The second line of input consists of **n** space-separated integer elements, forming the min heap sequence.

Output format :

The output prints the space-separated **n** integer values of the max heap array formed after the conversion.

Refer to the sample output for further format specifications.

Code constraints :

The test cases will fall under the following constraints:
1 ≤ n ≤ 30
1 ≤ array elements ≤ 35

Sample test cases :

Input 1 :

```
10
3 5 9 6 8 20 10 12 18 9
```

Output 1 :

```
20 18 10 12 9 9 3 5 6 8
```

Input 2 :

```
5
5 6 8 11 15
```

Output 2 :

```
15 11 8 5 6
```

```
#include <iostream>

using namespace std;


// to heapify a subtree with root at given index

void MaxHeapify(int arr[], int i, int N)

{

   int l = 2 * i + 1;

   int r = 2 * i + 2;

   int largest = i;


   if (l < N && arr[l] > arr[i])

      largest = l;
```

```
    if (r < N && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        MaxHeapify(arr, largest, N);
    }
}

// This function basically builds max heap
void convertMaxHeap(int arr[], int N)
{
    // Start from bottommost and rightmost internal node and heapify all internal nodes in bottom up way
    for (int i = (N - 2) / 2; i >= 0; --i)
        MaxHeapify(arr, i, N);
}



void printArray(int* arr, int size)
{
    for (int i = 0; i < size; ++i)
        cout << arr[i] << " ";
    cout<<endl;
}

int main()
{
    int arrayCount;
    cin>>arrayCount;
    int arr[arrayCount];
```

```
    for (int i=0;i<arrayCount;i++){
        cin>>arr[i];
    }

    convertMaxHeap(arr, arrayCount);

    printArray(arr, arrayCount);

    return 0;
}
```

Problem Statement

Isabella is working on a scheduling algorithm for a job queue. Each job has an associated priority value, and she needs to create a function to insert a new job, represented by an integer priority, into a **min heap**.

Write a code to help her insert a new job (an integer representing priority) into a min heap.

Input format :

The input consists of a series of space-separated integers, each representing the priority of a new job to be inserted into the min heap.

Output format :

The output displays the priorities of the jobs after each insertion into the min heap. The priorities are separated by spaces.

Code constraints :

The maximum number of jobs that can be inserted is 100, as the min heap has a maximum capacity of 100.

Sample test cases :

Input 1 :

```
5 3 8 2 1 6 4 7 9
```

Output 1 :

```
1 2 4 5 3 8 6 7 9
```

Input 2 :

```
10 15 20 17 25 30 40 35 50 45
```

Output 2 :

```
10 15 20 17 25 30 40 35 50 45
```

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

struct Job {
    int priority;
};

void swap(Job& a, Job& b) {
    Job temp = a;
    a = b;
    b = temp;
}

// Function to insert a new job into the min heap based on priority
void insertJob(Job heap[], int& heapSize, int newJobPriority) {
    heapSize++;

    int i = heapSize - 1;
    heap[i].priority = newJobPriority;

    while (i > 0 && heap[(i - 1) / 2].priority > heap[i].priority) {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printHeap(Job heap[], int heapSize) {
    for (int i = 0; i < heapSize; i++) {
        cout << heap[i].priority << " ";
```

```
    }
    cout << endl;
}

int main() {
    Job minHeap[100];
    int heapSize = 0;

    while (true) {
        int newJobPriority;
        if (!(cin >> newJobPriority)) {
            break;
        }

        insertJob(minHeap, heapSize, newJobPriority);
    }

    printHeap(minHeap, heapSize);

    return 0;
}
```

Problem Statement

You have been tasked with developing a game titled "The Treasure Hunt." In this game, the developers have chosen to implement a distinctive approach to organizing the treasure hunt by utilizing a min-heap data structure.

Each treasure is denoted by an integer value, and your assignment is to design a program that constructs a min-heap from the provided array of integers.

Input format :

The first line of the input consists of an integer **n**, representing the total number of treasures.

The second line of the input consists of **n** space-separated integer values representing the treasures.

Output format :

The output prints the given integer array of treasure values into a min-heap data structure. The values are space-separated.

Refer to the sample output for the exact format specifications.

Code constraints :

The test cases will fall under the following constraints:

1≤ n ≤ 30

1≤ values of the treasure ≤ 50

Sample test cases :

Input 1 :

```
5
2 4 1 5 9
```

Output 1 :

```
1 4 2 5 9
```

Input 2 :

```
7
8 7 6 4 5 1 2
```

Output 2 :

```
1 4 2 7 5 6 8
```

```
#include <iostream>

using namespace std;


void min_heap(int *a, int m, int n){

  int j, t;

  t= a[m];

  j = 2 * m;

  while (j <= n) {

    if (j < n && a[j+1] < a[j])

      j = j + 1;

    if (t < a[j])

      break;

    else if (t >= a[j]) {

      a[j/2] = a[j];
```

```
        j = 2 * j;
      }
   }
   a[j/2] = t;
   return;
}
void build_minheap(int *a, int n) {
   int k;
   for(k = n/2; k >= 1; k--) {
      min_heap(a,k,n);
   }
}
int main() {
   int n, i;
   cin>>n;

   int a[n];
   for (i = 1; i <= n; i++) {
     cin>>a[i];
   }
   build_minheap(a, n);
   for (i = 1; i <= n; i++) {
      cout<<a[i]<<" ";
   }
}
```

Problem Statement

Emily is developing a simple chat application where messages are represented as single characters. She needs to create a program that allows her to insert new messages into a **max heap**.

Write a program to assist Emily in inserting new chat messages (each represented by a single character) into a max heap.After inserting each message into the max heap, the program should display the current state of the max heap.

Input format :

The input consists of a series of space-separated characters, with each character representing a chat message.

Output format :

The output displays the characters of the messages in the max heap, ordered by their ASCII values.

Code constraints :

The maximum number of messages that can be inserted is 100, as the max heap has a maximum capacity of 100.

Sample test cases :

Input 1 :

```
c b a d e
```

Output 1 :

```
e d a b c
```

Input 2 :

```
z y x
```

Output 2 :

```
z y x
```

```
#include <iostream>

using namespace std;


struct Message {

   char content;

};


void swap(Message& a, Message& b) {

   Message temp = a;

   a = b;

   b = temp;
```

```
}

// Function to insert a new message into the max heap
void insertMessage(Message heap[], int& heapSize, Message newMessage) {
    heapSize++;

    int i = heapSize - 1;
    heap[i] = newMessage;

    while (i > 0 && heap[(i - 1) / 2].content < heap[i].content) {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printHeap(Message heap[], int heapSize) {
    for (int i = 0; i < heapSize; i++) {
        cout << heap[i].content << " ";
    }
    cout << endl;
}

int main() {
    Message maxHeap[100];  // Assuming a maximum heap size of 100
    int heapSize = 0;

    while (true) {
        Message newMessage;
        if (!(cin >> newMessage.content)) {
```

```
            break;
        }

        insertMessage(maxHeap, heapSize, newMessage);
    }
    printHeap(maxHeap, heapSize);

    return 0;
}
```

Problem Statement

Benjamin is developing a print job management system where users submit print jobs with varying importance, and he needs to insert new print jobs into a binary min-heap.

Write the code to help him insert the print jobs into a binary min-heap.

Input format :

The input consists of a series of integers, representing the importance of print jobs, that are to be inserted into the heap.

Output format :

The output prints the binary min-heap, after inserting all the print jobs.

Code constraints :

The maximum heap size is defined by maxHeapSize (here set to 100).

Sample test cases :

Input 1 :

```
5 3 8 1 4
```

Output 1 :

```
1 3 8 5 4
```

Input 2 :

```
10 7 2 9 6
```

Output 2 :

```
2 6 7 10 9
```

```
#include <iostream>

using namespace std;
```

```
struct PrintJob {
    int importance;
};

void swap(struct PrintJob* a, struct PrintJob* b) {
    struct PrintJob temp = *a;
    *a = *b;
    *b = temp;
}

void insertPrintJob(struct PrintJob heap[], int* heapSize, struct PrintJob newPrintJob) {
    (*heapSize)++;

    int i = (*heapSize) - 1;
    heap[i] = newPrintJob;

    while (i > 0 && heap[(i - 1) / 2].importance > heap[i].importance) {
        swap(&heap[i], &heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printHeap(struct PrintJob heap[], int heapSize) {
    for (int i = 0; i < heapSize; i++) {
        cout << heap[i].importance << " ";
    }
    cout << endl;
}
```

```
int main() {
    struct PrintJob binaryHeap[100];
    int heapSize = 0;

    while (1) {
        struct PrintJob newPrintJob;
        if (!(cin >> newPrintJob.importance)) {
            break;
        }

        insertPrintJob(binaryHeap, &heapSize, newPrintJob);
    }

    printHeap(binaryHeap, heapSize);

    return 0;
}
```

Problem Statement

Krish wants to sort a list of integers using the heap sort algorithm. He needs your help to implement this algorithm and is looking for a program that can take a list of integers as input, sort them in descending order using heap sort with max heap, and then print the sorted list.

Input format :

The first line of input consists of an integer **n**, the number of elements in the array.
The second line of input consists of **n** elements, separated by a space.

Output format :

The output displays a single line containing n space-separated integers, which represent the sorted list of integers in descending order.

Code constraints :

The test cases will fall under the following constraints:
1 <= n <= 10
1 <= elements <= 100

Sample test cases :

Input 1 :

```
5
47 78 84 50 94
```

Output 1 :

```
94 84 78 50 47
```

Input 2 :

```
8
65 95 74 12 36 84 20 67
```

Output 2 :

```
95 84 74 67 65 36 20 12
```

```
#include <iostream>

using namespace std;


void heapify(int arr[], int n, int root) {
   int largest = root;
   int left = 2 * root + 1;
   int right = 2 * root + 2;


   if (left < n && arr[left] > arr[largest])
      largest = left;


   if (right < n && arr[right] > arr[largest])
      largest = right;


   if (largest != root) {
      swap(arr[root], arr[largest]);
      heapify(arr, n, largest);
   }
}


void buildMaxHeap(int arr[], int n) {
   for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);
}

void heapSort(int arr[], int n) {
    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    int n;
    cin >> n;

    int arr[n];

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    heapSort(arr, n);

    for (int i = n - 1; i >= 0; i--) {
        cout << arr[i] << " ";
    }

    return 0;
```

}

Problem Statement

Rohith is an engineer who loves to work with lights. He has a collection of different types of lights, and he wants to arrange them in ascending order based on their brightness. To accomplish this, he decided to write a program that uses Max Heap to sort the lights.

Write a program to help Rohith sort his lights in ascending order of brightness using Heap Sort.

Input format :

The first element in the first line denotes the number n_0. After that, n_0 space-separated integer represents the brightness level.

The first element in the second line denotes the number n_1. After that, n_1 space-separated integers represent the brightness level.

The first element in the third line denotes the number n_3. After that, n_3 space-separated integers represent the brightness level.

Refer to the sample input for a better understanding.

Output format :

The output prints the brightness levels of all lights in ascending order, separated by a space.

Code constraints :

$1 \leq (n_0 + n_1 + n_2) \leq 20$

$0 \leq$ Brightness level ≤ 100

Sample test cases :

Input 1 :

```
1 5
3 9 10 13
2 15 78
```

Output 1 :

```
5 9 10 13 15 78
```

Input 2 :

```
1 10
1 10
1 10
```

Output 2 :

```
10 10 10
```

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

```
void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   if (l < n && arr[l] > arr[largest])
      largest = l;

   if (r < n && arr[r] > arr[largest])
      largest = r;

   if (largest != i) {
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;
      heapify(arr, n, largest);
   }
}

void buildMaxHeap(int arr[], int n)
{
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);
}

void heapSort(int arr[], int n) {
   buildMaxHeap(arr, n);
```

```
    for (int i = n - 1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

int main() {
    int no, n1, n2;
    scanf("%d", &no);
    int arr[no];
    for (int i = 0; i < no; i++)
        scanf("%d", &arr[i]);
    scanf("%d", &n1);
    for (int i = no; i < no + n1; i++)
        scanf("%d", &arr[i]);
    scanf("%d", &n2);
    for (int i = no + n1; i < no + n1 + n2; i++)
        scanf("%d", &arr[i]);

    int totalLights = no + n1 + n2;
    heapSort(arr, totalLights);

    for (int i = 0; i < totalLights; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Problem Statement

Dustin is learning about data structures and algorithms in his computer science class. Recently, he came across the concept of a Max Heap. To practice his understanding, he decided to implement a Max Heap and test it using a program.

Write a program that performs the following tasks:

1. Create a Max Heap from a given list of integers.
2. Remove and print the maximum element from the Max Heap.
3. After rearranging the heap, print the current order of it.

Dustin needs your help to design this program. Help him solve the program.

Input format :

The first line consists of an integer n, representing the number of elements in the list.
The next line contains n space-separated integers where each integer represents an element of the list.

Output format :

If the input is invalid (negative or zero), the output prints "Invalid number of elements."
If the input is valid,

- The first line prints the maximum element removed from the Max Heap.
- After rearranging the heap, the second line prints the elements of the Max Heap in their current order.

Refer to the sample output for a better understanding.

Code constraints :

$1 \leq n \leq 10$

$1 \leq$ Elements ≤ 100

Sample test cases :

Input 1 :

```
6
30 15 40 10 50 25
```

Output 1 :

```
50
40 30 25 10 15
```

Input 2 :

```
-9
```

Output 2 :

```
Invalid number of elements.
```

Input 3 :

```
0
```

Output 3 :

```
Invalid number of elements.
```

```
#include <iostream>

using namespace std;

void maxHeapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        maxHeapify(arr, n, largest);
    }
}

int removeMax(int arr[], int& n) {
    if (n <= 0)
        return -1;

    int maxElement = arr[0];

    arr[0] = arr[n - 1];
    n--;
```

```
    maxHeapify(arr, n, 0);

    return maxElement;
}

int main() {
    int* heap = nullptr;
    int n;

    cin >> n;

    if (n <= 0) {
        cout << "Invalid number of elements." << endl;
        return 0;
    }

    heap = new int[n];

    for (int i = 0; i < n; i++) {
        cin >> heap[i];
    }

    if (n > 0) {
        for (int i = n / 2 - 1; i >= 0; i--) {
            maxHeapify(heap, n, i);
        }

        int maxElement = removeMax(heap, n);
```

```
        cout << maxElement << endl;
    }

    for (int i = 0; i < n; i++)
        cout << heap[i] << " ";
    cout << endl;

    delete[] heap;

    return 0;
}
```

Problem Statement

Given a group of students with varying scores, implement a program that constructs a max-heap to represent their scores. Serve an award to the student with the highest score and print the score of the student who receives the award.

Input format :

The first line consists of an integer n representing the number of students.

The second line consists of n space-separated integers, each representing the score of a student.

Output format :

The output prints the highest score of the student who receives the award.

Code constraints :

In this scenario, the test cases fall under the following constraints:

1 <= n <= 10

1 <= score <= 100

Sample test cases :

Input 1 :

```
8
95 88 77 92 84 60 70 99
```

Output 1 :

```
99
```

Input 2 :

```
1
100
```

Output 2 :

```
100
```

Input 3 :

```
4
80 80 80 80
```

Output 3 :

```
80
```

```
#include <iostream>

using namespace std;


void maxHeapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;


   if (left < n && arr[left] > arr[largest])
      largest = left;


   if (right < n && arr[right] > arr[largest])
      largest = right;


   if (largest != i) {
      swap(arr[i], arr[largest]);
      maxHeapify(arr, n, largest);
   }
}


int removeHighestScore(int arr[], int& n) {
   if (n <= 0)
      return -1;


   int highestScore = arr[0];
```

```
    arr[0] = arr[n - 1];
    n--;

    maxHeapify(arr, n, 0);

    return highestScore;
}

int main() {
    int* studentScores = nullptr;
    int n = 0;

    cin >> n;
    studentScores = new int[n];

    for (int i = 0; i < n; i++) {
        cin >> studentScores[i];
    }

    for (int i = n / 2 - 1; i >= 0; i--) {
        maxHeapify(studentScores, n, i);
    }

    int highestScore = removeHighestScore(studentScores, n);
    cout << highestScore << endl;
    delete[] studentScores;
    return 0;
}
```

Problem Statement

Priya has a garden with different types of flowers, and they are arranged in a max heap based on their height, with the tallest flower at the top. Write a program that helps Priya delete the tallest flower from the garden for pruning and display the remaining heights of the flowers in the garden.

Input format :

The first consists of an integer n representing the number of flowers in the garden.
The second line consists of n space-separated integers representing the heights of each flower.

Output format :

After pruning the tallest flower, the program displays the heights of the remaining flowers separated by spaces.

Code constraints :

In this scenario, the test cases fall under the following constraints:

$1 \leq n \leq 100$

$1 \leq \text{Heights} \leq 100$

Sample test cases :

Input 1 :

```
7
90 85 70 60 75 80 95
```

Output 1 :

```
90 85 80 60 75 70
```

Input 2 :

```
5
80 65 75 90 70
```

Output 2 :

```
80 70 75 65
```

```
#include <iostream>

using namespace std;


void swap(int &x, int &y) {

   int temp = x;

   x = y;

   y = temp;

}


void heapifyDown(int *heap, int n, int i) {
```

```
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && heap[left] > heap[largest])
        largest = left;

    if (right < n && heap[right] > heap[largest])
        largest = right;

    if (largest != i) {
        swap(heap[i], heap[largest]);
        heapifyDown(heap, n, largest);
    }
}

void deleteTallestFlower(int *heap, int &n) {
    swap(heap[0], heap[n - 1]);
    n--;

    heapifyDown(heap, n, 0);
}

int main() {
    int *heap;
    int n;
    cin >> n;
    heap = new int[n];
```

```
    for (int i = 0; i < n; i++) {
        cin >> heap[i];
    }

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapifyDown(heap, n, i);
    }

    deleteTallestFlower(heap, n);

    for (int i = 0; i < n; i++) {
        cout << heap[i] << " ";
    }
    cout << endl;
    delete[] heap;
    return 0;
}
```

Problem Statement

Uma is working on a project where she needs to sort a list of strings lexicographically. She recently learned about Heap Sort and decided to implement it to solve her problem. However, she wants to sort the strings in ascending order using a Min Heap.

Write a program that takes a list of strings as input and sorts them lexicographically in ascending order using a Min Heap.

Input format :

The first line of input consists of an integer n, the number of strings to be sorted.
The second line of input consists of the strings. Each string contains only lowercase and uppercase letters, separated by a space.

Output format :

The output displays the sorted list of strings in lexicographically ascending order, separated by a space.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq n \leq 10$

The maximum length of 100 characters.

Sample test cases :

Input 1 :

```
7
Morlee
Zabrina
Domeniga
Alexandr
Odelia
Laura
Sibyl
```

Output 1 :

```
Alexandr Domeniga Laura Morlee Odelia Sibyl Zabrina
```

Input 2 :

```
5
Foss
Cele
Dredi
Bria
Eve
```

Output 2 :

```
Bria Cele Dredi Eve Foss
```

```
#include <iostream>

using namespace std;


#define MAX_STR_LEN 100

#define MAX_NUM_STR 100


int compareStrings (char* str1, char* str2) {

   int i = 0;

   while (str1[i] != '\0' && str2[i] != '\0') {

      if (str1[i] < str2[i])

         return -1;

      else if (str1[i] > str2[i])

         return 1;

      i++;
```

```
    }
    if (str1[i] == '\0' && str2[i] == '\0')
        return 0;
    else if (str1[i] == '\0')
        return -1;
    else
        return 1;
}

void copyString(char* dest, char* src) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
}

void MinHeapify(char arr[][MAX_STR_LEN], int n, int i) {
    int smallest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && compareStrings(arr[l], arr[smallest]) < 0)
        smallest = l;

    if (r < n && compareStrings(arr[r], arr[smallest]) < 0)
        smallest = r;
```

```
    if (smallest != i) {
        char temp[MAX_STR_LEN];
        copyString(temp, arr[i]);
        copyString(arr[i], arr[smallest]);
        copyString(arr[smallest], temp);
        MinHeapify(arr, n, smallest);
    }
}

void MinHeapSort(char arr[][MAX_STR_LEN], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        MinHeapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        cout << arr[0] << " ";
        char temp[MAX_STR_LEN];
        copyString(temp, arr[0]);
        copyString(arr[0], arr[i]);
        copyString(arr[i], temp);
        MinHeapify(arr, i, 0);
    }
}

int main() {
    int n;
    cin >> n;

    char arr[MAX_NUM_STR][MAX_STR_LEN];
    for (int i = 0; i < n; i++) {
```

```
        cin >> arr[i];
    }

    MinHeapSort(arr, n);

    return 0;
}
```

Problem Statement

Laxmi is working with a set of integers and wants to filter out even numbers from her data. She's interested in creating a program that can build a max-heap from an array of integers and then efficiently remove all even elements from the heap. Can you help Laxmi by designing a program that accomplishes this task?

Your task is to create a program that reads an array of integers, builds a max-heap from it, and then deletes all even elements from the max-heap, and display the resulting max-heap.

Input format :

The first line of input consists of an integer **n,** the number of elements in the array.
The second line of input consists of **n** elements separated by spaces.

Output format :

The output displays the resulting max-heap after deleting all even elements.

Refer to the sample output for formatting specifications.

Code constraints :

The test cases will fall under the following constraints:
1 <= n <= 10
1 <= elements <= 100

Sample test cases :

Input 1 :

```
8
50 40 30 45 35 25 15 5
```

Output 1 :

```
45 35 25 5 15
```

Input 2 :

```
5
10 20 37 40 50
```

Output 2 :

```
37
```

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void buildMaxHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
}
void deleteEvenElements(int arr[], int& n) {
    int i, j;
    int evenCount = 0;

    for (i = 0; i < n; i++) {
```

```
        if (arr[i] % 2 == 0) {
            evenCount++;
        }
    }
    if (evenCount == 0) {
        return;
    }

    for (i = 0; i < n; i++) {
        if (arr[i] % 2 == 0) {
            arr[i] = arr[n - 1];
            n--;

            buildMaxHeap(arr, n);
            i--;
        }
    }
}

void displayHeap(int arr[], int n) {

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
}

int main() {
    int n;
    cin >> n;
```

```
    int arr[n];

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    buildMaxHeap(arr, n);

    deleteEvenElements(arr, n);

    displayHeap(arr, n);

    return 0;
}
```

Problem Statement

Vijay is fascinated by data structures and algorithms, particularly heap data structures. He wants to experiment with a program that implements a max heap and performs heap operations. He wishes to understand how deleting the root element from a max heap works.

Write a program that defines a max heap and implements the deletion of the root element. The program should receive an array representing the max heap, its size, and delete the root element. After deletion, the program should print the updated array.

Input format :

The first line of input contains an integer 'n', the size of the array representing the max heap.

The second line contains 'n' space-separated integers, representing the elements of the max heap.

Output format :

The output prints the updated max heap after deleting the root element, separated by a space.

Code constraints :

The test cases will under the following constraints:

$1 \leq n \leq 10$

The input array can have duplicate elements.

Sample test cases :

Input 1 :

```
5
23 25 2 12 52
```

Output 1 :

```
25 23 2 12
```

Input 2 :

```
5
10 5 3 2 4
```

Output 2 :

```
5 4 3 2
```

```
#include <iostream>

using namespace std;


#define MAX_SIZE 100


void heapify(int arr[], int n, int i)

{

   int largest = i;

   int l = 2 * i + 1;

   int r = 2 * i + 2;


   if (l < n && arr[l] > arr[largest])

      largest = l;


   if (r < n && arr[r] > arr[largest])

      largest = r;


   if (largest != i)

   {
```

```
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}

void buildMaxHeap(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}

void deleteRoot(int arr[], int &n)
{
    if (n <= 0)
        return;

    int lastElement = arr[n - 1];
    arr[0] = lastElement;
    n = n - 1;
    heapify(arr, n, 0);
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
}
```

```
int main()
{
   int n;
   cin >> n;

   int arr[MAX_SIZE];

   for (int i = 0; i < n; ++i)
      cin >> arr[i];

   buildMaxHeap(arr, n);
   deleteRoot(arr, n);
   printArray(arr, n);

   return 0;
}
```

Problem Statement

Sophie, a computer science student, is learning about Binary Search Trees (BSTs) and wants to create a program to practice working with BSTs.

She needs a program that allows her to insert a node with a specific value into a BST and find the number of nodes in the tree that have the same value as a given reference value.

Input format :

The first line of input consists of the number of nodes in the BST, **N.**

The second line of input consists of N space-separated integers representing the values of the BST nodes in the order they were inserted.

The third line of input consists of an integer, **K,** representing the specific value to insert into the BST.

The fourth line of input consists of an integer, **X,** representing the reference value for which Sophie wants to find the number of matching nodes.

Output format :

- The first line of output displays the values of the BST in ascending order after inserting the specific value (K).
- The second line of output displays the count of nodes in the BST that have the same value as the reference value (X).

Refer to the sample output for the formatting specifications.

Code constraints :

The given test cases will fall under the following constraints:

$1 \leq n \leq 10$

1 ≤ value of nodes ≤ 1000

$1 \leq K \leq 1000$

$1 \leq X \leq 1000$

Sample test cases :

Input 1 :

```
5
10 5 20 3 8
15
5
```

Output 1 :

```
3 5 8 10 15 20
1
```

Input 2 :

```
7
50 30 70 20 40 60 80
40
40
```

Output 2 :

```
20 30 40 40 50 60 70 80
2
```

Input 3 :

```
3
10 15 5
5
5
```

Output 3 :

```
5 5 10 15
2
```

```
#include <iostream>

using namespace std;


struct TreeNode {

   int data;
```

```
    TreeNode* left;
    TreeNode* right;
    TreeNode* next;
};

TreeNode* createNode(int key) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode)); // Use malloc for memory allocation
    newNode->data = key;
    newNode->left = nullptr;
    newNode->right = nullptr;
    newNode->next = nullptr;
    return newNode;
}

TreeNode* insert(TreeNode* root, int key) {
    if (root == nullptr) return createNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);
    else {
        TreeNode* currentNode = root;
        while (currentNode->next != nullptr) {
            currentNode = currentNode->next;
        }
        currentNode->next = createNode(key);
    }
    return root;
}
```

```
int countNodesWithValue(TreeNode* root, int value) {
    if (root == nullptr) return 0;
    int count = countNodesWithValue(root->left, value) + countNodesWithValue(root->right, value);

    TreeNode* currentNode = root;
    while (currentNode != nullptr) {
        if (currentNode->data == value) {
            count++;
        }
        currentNode = currentNode->next;
    }

    return count;
}

void printInOrder(TreeNode* root) {
    if (root != nullptr) {
        printInOrder(root->left);

        cout << root->data << " ";

        TreeNode* currentNode = root->next;
        while (currentNode != nullptr) {
            cout << currentNode->data << " ";
            currentNode = currentNode->next;
        }

        printInOrder(root->right);
```

```
    }
}

int main() {
    int N, K, X;
    cin >> N;

    TreeNode* root = nullptr;

    for (int i = 0; i < N; i++) {
        int key;
        cin >> key;
        root = insert(root, key);
    }

    cin >> K >> X;
    root = insert(root, K);

    printInOrder(root);
    cout << endl;
    int count = countNodesWithValue(root, X);
    cout << count << endl;

    return 0;
}
```

Problem Statement

Vijay, a computer science student, is learning about binary search trees (BSTs) and wants to create a program that prints the ancestors of a specific node in a BST. He needs help to solve a problem.

Example

Input

8

40 20 60 10 30 46 50 23

23

Output

30 20 40

Explanation

40
/ \
20 60
/ \ \
10 30 46
/ /
23 50

The ancestors of the node with value 23 are the nodes in the path from the root to the node with value 23. In this case, the ancestors are 30, 20, and 40.

Input format :

The first line of the input consists of an integer N representing the size of the array.
The second line of the input consists of N space-separated integer values.
The third line of the input consists of an integer M representing the Key value for finding the ancestors.

Output format :

The output displays the values of the ancestors of the specified node in preorder traversal, separated by a space.

Refer to the sample output for the formatting specifications.

Code constraints :

The given test cases will fall under the following constraints:

1 <= N <= 20

1 <= values <= 1000

1 <= M <= 1000

Sample test cases :

Input 1 :

```
3
30 20 40
40
```

Output 1 :

```
30
```

Input 2 :

```
8
40 20 60 10 30 46 50 23
23
```

Output 2 :

```
30 20 40
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;
};

Node* createNode(int data) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->data = data;
   newNode->left = nullptr;
   newNode->right = nullptr;
   return newNode;
}

Node* insert(Node* root, int data) {
   if (root == nullptr) {
      return createNode(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
```

```
    }
    else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

int search(Node* root, int key) {
    if (root == nullptr) {
        return 0; // Node not found
    }
    if (root->data == key) {
        return 1; // Node found
    }
    if (key < root->data) {
        if (search(root->left, key)) {
            cout << root->data << " ";
            return 1;
        }
    }
    else {
        if (search(root->right, key)) {
            cout << root->data << " ";
            return 1;
        }
    }
    return 0; // Node not found
}
```

```
void printAncestors(Node* root, int key) {
    search(root, key);
}

void preorder(Node* root) {
    if (root == nullptr) {
        return;
    }
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

int main() {
    Node* root = nullptr;
    int n, data, key;

    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> data;
        root = insert(root, data);
    }

    cin >> key;

    printAncestors(root, key);

    return 0;
```

}

Problem Statement

Banu is studying binary search trees (BST) in her computer science class. She needs to write a program to manage BST nodes, including insertion and deletion. Her task is to delete nodes with values greater than a given threshold from a BST.

You are required to help Banu by implementing a program for this task.

Input format :

The first line of input consists of an integer n, the number of nodes in the initial BST.
The second line of input consists of n space-separated integers, representing the values of the nodes in the initial BST.
The third line of input consists of an integer v, the threshold value for node deletion.

Output format :

The output displays the values of the nodes remaining in the BST after deleting nodes with values greater than the given threshold, in an in-order traversal order.

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq n \leq 10$
$1 \leq$ value of nodes ≤ 1000
$1 \leq v \leq 1000$

Sample test cases :

Input 1 :

```
6
5 4 3 6 2 9
7
```

Output 1 :

```
2 3 4 5 6
```

Input 2 :

```
3
20 48 24
24
```

Output 2 :

```
20
```

```
#include <iostream>

struct Node {
```

```
    int data;
    Node* left;
    Node* right;
};

Node* crNode(int d) {
    Node* nN = (Node*)malloc(sizeof(Node));
    nN->data = d;
    nN->left = nN->right = nullptr;
    return nN;
}

Node* insrt(Node* r, int d) {
    if (r == nullptr) {
        return crNode(d);
    }
    if (d < r->data) {
        r->left = insrt(r->left, d);
    } else if (d > r->data) {
        r->right = insrt(r->right, d);
    }
    return r;
}

Node* dltGt(Node* r, int v) {
    if (r == nullptr) {
        return nullptr;
    }
    r->left = dltGt(r->left, v);
```

```
    r->right = dltGt(r->right, v);

    if (r->data > v) {
        Node* t = r;
        r = nullptr;
        free(t);
    }
    return r;
}

void iOT(Node* r) {
    if (r != nullptr) {
        iOT(r->left);
        cout << r->data << " ";
        iOT(r->right);
    }
}

int main() {
    Node* r = nullptr;
    int n, d, v;

    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> d;
        r = insrt(r, d);
    }
```

```
    cin >> v;

    r = dltGt(r, v);

    iOT(r);

    return 0;
}
```

Problem Statement

John is a computer science student who is currently studying Binary Search Trees (BSTs). He is enthusiastic about implementing various operations on BSTs. His current goal is to create a program that allows users to efficiently delete the node with the minimum value from a BST.

Can you help John achieve his goal by implementing a program to delete the node with the minimum value from a given BST?

Input format :

The first line of input consists of an integer **N,** representing the number of nodes in the BST.

The second line consists of **N** space-separated integers, representing the values of the BST nodes in the order they were inserted.

Output format :

The program should display a single line containing the values of the BST nodes in ascending order after deleting the node with the minimum value.

If the tree contains only one element, the output should be "(Empty tree)".

Code constraints :

The input values are unique integers.

Sample test cases :

Input 1 :

```
7
5 3 8 2 4 7 10
```

Output 1 :

```
3 4 5 7 8 10
```

Input 2 :

```
5
10 15 5 7 20
```

Output 2 :

```
7 10 15 20
```

Input 3 :

```
3
7 5 10
```

Output 3 :

```
7 10
```

Input 4 :

```
1
42
```

Output 4 :

```
(Empty tree)
```

```
#include <iostream>


// Define a structure for a binary tree node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val) : data(val), left(NULL), right(NULL) {}
};


// Function to insert a new key into a BST
TreeNode* insert(TreeNode* root, int key) {
    if (root == NULL) return new TreeNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);
    return root;
}


// Function to delete the node with the minimum value in the BST
TreeNode* deleteMinNode(TreeNode* root) {
    if (root == NULL) return NULL;
```

```
    if (root->left == NULL) {
        TreeNode* temp = root->right;
        delete root;
        return temp;
    }
    root->left = deleteMinNode(root->left);
    return root;
}

// Function to print the values of the BST nodes
void printValues(TreeNode* root) {
    if (root != NULL) {
        printValues(root->left);
        std::cout << root->data << " ";
        printValues(root->right);
    }
}

int main() {
    int N;
    std::cin >> N;

    TreeNode* root = NULL;

    for (int i = 0; i < N; i++) {
        int key;
        std::cin >> key;
        root = insert(root, key);
    }
```

```
    root = deleteMinNode(root);


    if (root == NULL) {
        std::cout << "(Empty tree)" << std::endl;
    } else {
        printValues(root);
        std::cout << std::endl;
    }


    return 0;
}
```

Problem Statement

You are tasked with writing a program to work with binary search trees (BST). In this program, you will create, modify, and display a BST.

Your task is to implement a program that performs the following operations:

1. Create a binary search tree (BST) by inserting a series of integer values.
2. Find and delete the node with the maximum value in the BST.
3. Display the BST after the deletion operation using an in-order traversal.

The provided code gives you a starting point with the structure and essential functions for handling the BST. Your task is to complete the missing parts and implement the specified functionality.

Input format :

The first line input consists of an integer **N,** representing the number of nodes to be inserted into the binary search tree (BST).

The second line consists of **N** integers, separated by spaces, representing the values to be inserted into the BST.

Output format :

The first line of output prints the value of the node with the maximum value that will be deleted.

The second line consists of the values of the nodes in the BST after the maximum value node has been deleted using an in-order traversal.

Refer to the sample output for the exact text.

Code constraints :

$1 \leq N \leq 25$

Sample test cases :

Input 1 :

```
3
50 30 70
```

Output 1 :

```
Node with the maximum value to be deleted: 70
BST after deleting the maximum value node: 30 50
```

Input 2 :

```
6
45 98 67 84 35 17
```

Output 2 :

```
Node with the maximum value to be deleted: 98
BST after deleting the maximum value node: 17 35 45 67 84
```

```
#include <iostream>


struct Node {
   int data;
   Node* left;
   Node* right;
};


// Function to create a new node
Node* createNode(int data) {
   Node* newNode = new Node;
   newNode->data = data;
   newNode->left = newNode->right = nullptr;
   return newNode;
}


// Function to insert a new node into the BST
Node* insert(Node* root, int data) {
```

```
    if (root == nullptr)
        return createNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);

    return root;
}

// Function to find the node with the maximum value in the BST
Node* findMax(Node* root) {
    while (root->right != nullptr)
        root = root->right;
    return root;
}

// Function to delete the node with the maximum value from the BST
Node* deleteMax(Node* root) {
    if (root == nullptr)
        return root;

    if (root->right == nullptr) {
        Node* temp = root->left;
        delete root;
        return temp;
    }
```

```
    root->right = deleteMax(root->right);
    return root;
}

// Function to perform an in-order traversal of the BST
void inOrderTraversal(Node* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left);
        std::cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}

int main() {
    Node* root = nullptr;

    // Input: Number of nodes
    int n;
    std::cin >> n;

    for (int i = 0; i < n; i++) {
        int value;
        std::cin >> value;
        root = insert(root, value);
    }

    // Find and delete the node with the maximum value
    Node* maxNode = findMax(root);
    std::cout << "Node with the maximum value to be deleted: " << maxNode->data <<
std::endl;
```

```
    root = deleteMax(root);


    std::cout << "BST after deleting the maximum value node: ";
    inOrderTraversal(root);
    // std::cout << std::endl;


    return 0;
}
```

Problem Statement:

Meenu is developing a program to manage the hierarchical structure of employees within a company. The company's organizational structure is represented as a **binary search tree**, where each employee is assigned a unique employee ID, and employees are organized based on their supervisor-subordinate relationships.

The program should enable the creation of this hierarchy by accepting user input and then calculating the total number of employees.

Input format :

The first line contains the total number of employees, N.

The next N lines contain the employee IDs.

Output format :

The output displays a single integer representing the total number of nodes (employees) in the binary search tree (company hierarchy).

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

1 <= n <= 10

1 <= employee ID <= 100

Sample test cases :

Input 1 :

```
7
50
30
70
20
40
60
80
```

Output 1 :

```
7
```

Input 2 :

```
6
40
30
50
20
35
45
```

Output 2 :

```
6
```

```
#include <iostream>

using namespace std;


int n = 1;


struct node

{

   int data;

   node* left;

   node* right;

};


struct node* getNode(int data)

{

   node* newNode = (node*)malloc(sizeof(node));

   newNode->data = data;

   newNode->left = NULL;

   newNode->right = NULL;

   return newNode;

}


struct node* Insert(struct node* root, int data)
```

```
{
    if (root == NULL)
        return getNode(data);

    if (data < root->data)
        root->left = Insert(root->left, data);
    else if (data > root->data)
        root->right = Insert(root->right, data);

    return root;
}

int CountNodes(node* root)
{
    if (root == NULL)
        return 0;

    if (root->left != NULL)
    {
        n = n + 1;
        n = CountNodes(root->left);
    }
    if (root->right != NULL)
    {
        n = n + 1;
        n = CountNodes(root->right);
    }
    return n;
}
```

```
int main()
{
   node* root = NULL;
   int numNodes;

   cin >> numNodes;

   for (int i = 0; i < numNodes; i++) {
      int data;
      cin >> data;
      root = Insert(root, data);
   }

   cout << CountNodes(root) << endl;

   return 0;
}
```

Problem Statement

Uma wants to calculate the depth of a node in the given Binary Search Tree. The depth of a node is defined as the number of edges present in the path from the root node of a tree to that node.

Help her implement a program to calculate the depth of a node in the given BST.

Input format :

The first line consists of an integer, **n,** denoting the number of nodes in the BST.
The second line consists of n space-separated integers, representing the elements to be inserted into the BST.
The third line consists of an integer k representing the node for which depth needs to be calculated.

Output format :

The output prints the depth of the given node.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq n \leq 15$

$1 \leq \text{Elements} \leq 300$

Sample test cases :

Input 1 :

```
3
8 6 10
6
```

Output 1 :

```
1
```

Input 2 :

```
6
20 10 30 5 15 25
5
```

Output 2 :

```
2
```

```
// You are using GCC
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(int item) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
int findDepth(struct Node* root, int x) {
    if (root == NULL)
        return -1;

    int dist = -1;

    if ((root->data == x) || (dist = findDepth(root->left, x)) >= 0 || (dist = findDepth(root->right, x)) >= 0)
        return dist + 1;

    return dist;
}

int main() {
    struct Node* root = newNode(0);

    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        int value;
        scanf("%d", &value);
        if (i == 0) {
            root->data = value;
        } else {
            struct Node* current = root;
            while (1) {
                if (value < current->data) {
                    if (current->left == NULL) {
                        current->left = newNode(value);
```

```
                break;
            }
            current = current->left;
        } else if (value > current->data) {
            if (current->right == NULL) {
                current->right = newNode(value);
                break;
            }
            current = current->right;
        } else {
            break;
        }
    }
  }

  int k;
  scanf("%d", &k);
  printf("%d\n", findDepth(root, k));
  return 0;
}
```

Problem Statement:

Geetha is exploring the world of binary search trees (BSTs) and wants to create a program to find the sum of all nodes in a BST that are greater than a given target value.

She has implemented a program to accomplish this and needs a problem statement for her task.

Input format :

The first line of input consists of an integer, n, representing the number of nodes in the BST.
The second line of input consists of an integer, denoting the values to be inserted into the BST.
The last line of input consists of a single integer, target, which is the target value.

Output format :

The output displays a single integer, which is the sum of all nodes in the BST with values greater than the target value.

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

1 <= n <= 10

1 <= target <= 100

Sample test cases :

Input 1 :

```
7
10 20 40 4 60 70 3
60
```

Output 1 :

```
70
```

Input 2 :

```
5
10 3 10 50 60
4
```

Output 2 :

```
120
```

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* left;

   struct Node* right;

};


struct Node* create(int value) {
```

```
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = value;
    node->left = node->right = NULL;
    return node;
}

struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return create(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

int sumGtr(struct Node* root, int value) {
    if (root == NULL) {
        return 0;
    }

    if (root->data > value) {
        return root->data + sumGtr(root->left, value) + sumGtr(root->right, value);
    } else {
        return sumGtr(root->right, value);
```

```
    }
}

int main() {
    struct Node* root = NULL;
    int n, data, target;

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }

    scanf("%d", &target);

    int sum = sumGtr(root, target);
    printf("%d", sum);

    return 0;
}
```