Bharatiya Vidya Bhavan's

SARDAR PATEL INSTITUTE OF TECHNOLOGY

(Autonomous Institute Affiliated to University of Mumbai)

Munshi Nagar, Andheri (W), Mumbai – 400 058.

COMPS Department

Experiment	10
Aim	**Implement the HeapSort algorithm (Min/Max). The program should accept an ARRAY A as input. The output should be Sorted Array A.**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	6-11-24

Theory

A **Heap** is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.

A Heap is a special **Tree-based Data Structure** that has the following properties.

- It is a complete Complete Binary Tree.
- It either follows max heap or min heap property.

Max-Heap: The value of the root node must be the greatest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

Min-Heap: The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

Properties of Heap:

- The minimum or maximum element is always at the root of the heap, allowing constant-time access.
- The relationship between a parent node at index **'i'** and its children is given by the formulas: left child at index **2i+1** and right child at index **2i+2** for 0-based indexing of node numbers.
- As the tree is complete binary, all levels are filled except possibly the last level. And the last level is filled from left to right.
- When we insert an item, we insert it at the last available slot and then rearrange the nodes so that the heap property is maintained.
- When we remove an item, we swap root with the last node to make sure either the max or min item is removed. Then we rearrange the remaining nodes to ensure heap property (max or min)

Operations Supported by Heap:

Operations supported by **min – heap** and **max – heap** are same. The difference is just that min-heap contains minimum element at root of the tree and max – heap contains maximum element at the root of the tree.
Heapify: It is the process to rearrange the elements to maintain the property of heap data structure. It is done when root is removed (we replace root with the last node and then call heapify to ensure that heap property is maintained) or heap is built (we call heapify from the last internal node to root) to make sure that the heap property is maintained. This operation also takes **O(log n)** time.

- For **max-heap,** it makes sure the maximum element is the root of that binary tree and all descendants also follow the same property.
- For **min-heap,** it balances in such a way that the minimum element is the root and all descendants also follow the same property.

Insertion: If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the **heapify** operation so that it maintains the property of the heap. This operation also takes **O(log n)** time.

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

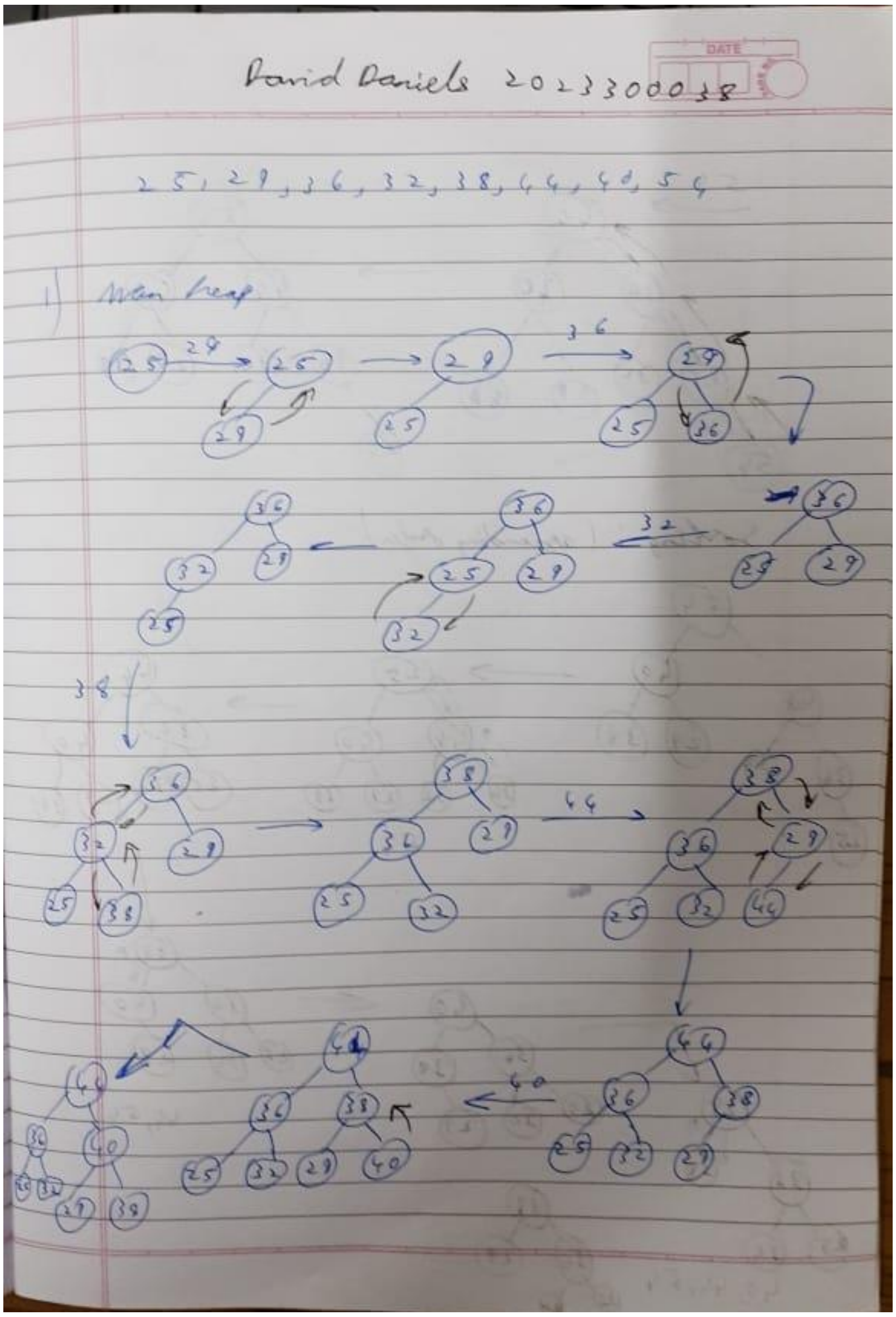
Advantages of Heap Data Structure

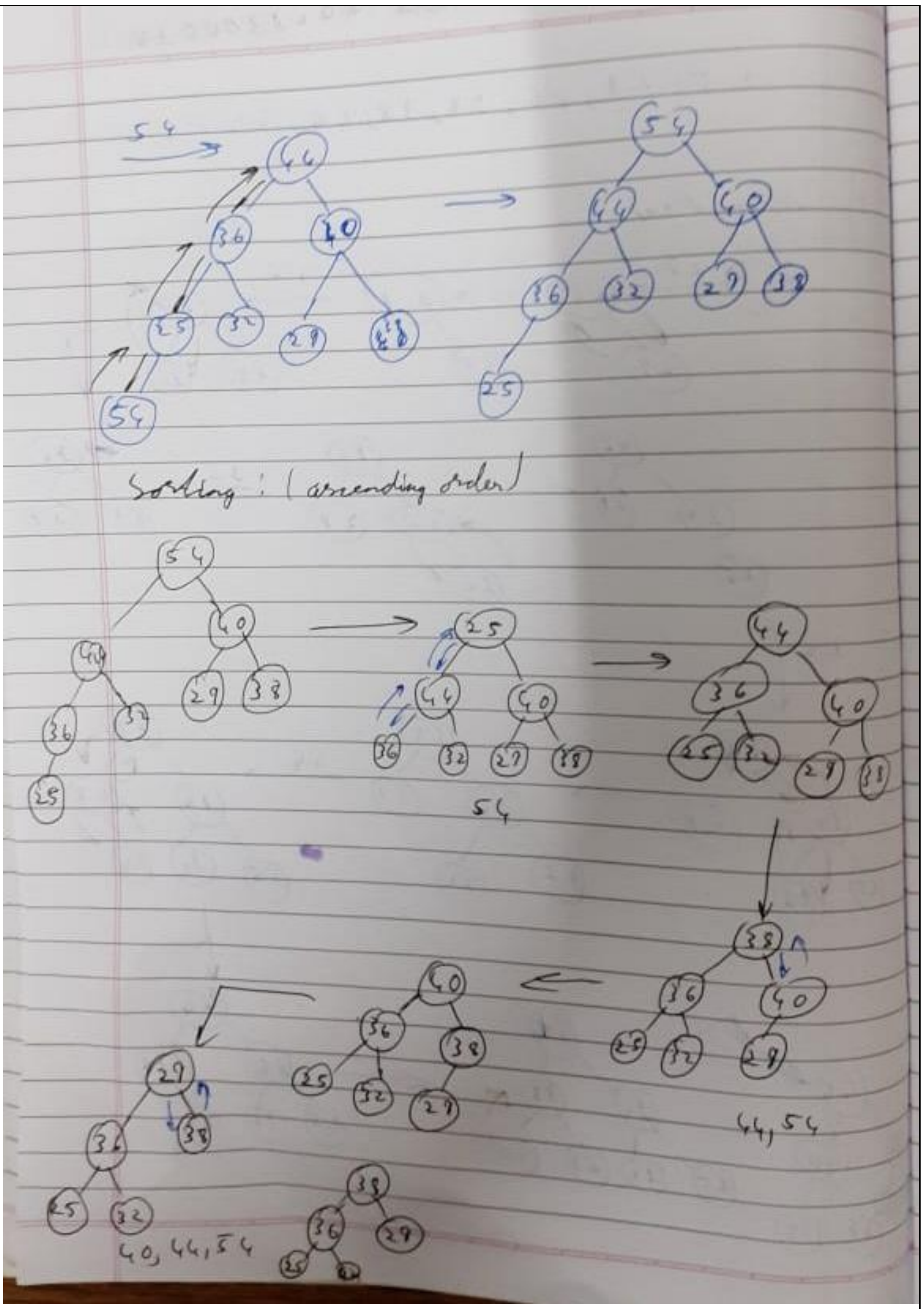
1. **Time Efficient**: Heaps have an average time complexity of O(log n) for inserting and deleting elements, making them efficient for large datasets. We can convert any array to a heap in O(n) time. The most important thing is, we can get the min or max in O(1) time
2. **Space Efficient** : A Heap tree is a complete binary tree, therefore can be stored in an array without wastage of space.
3. **Dynamic**: Heaps can be dynamically resized as elements are inserted or deleted, making them suitable for dynamic applications that require adding or removing elements in real-time.
4. **Priority-based:** Heaps allow elements to be processed based on priority, making them suitable for real-time applications, such as load balancing, medical applications, and stock market analysis.
5. **In-place**: Most of the applications of heap require in-place rearrangements of elements. For example HeapSort.

Disadvantages of Heap Data Structure:

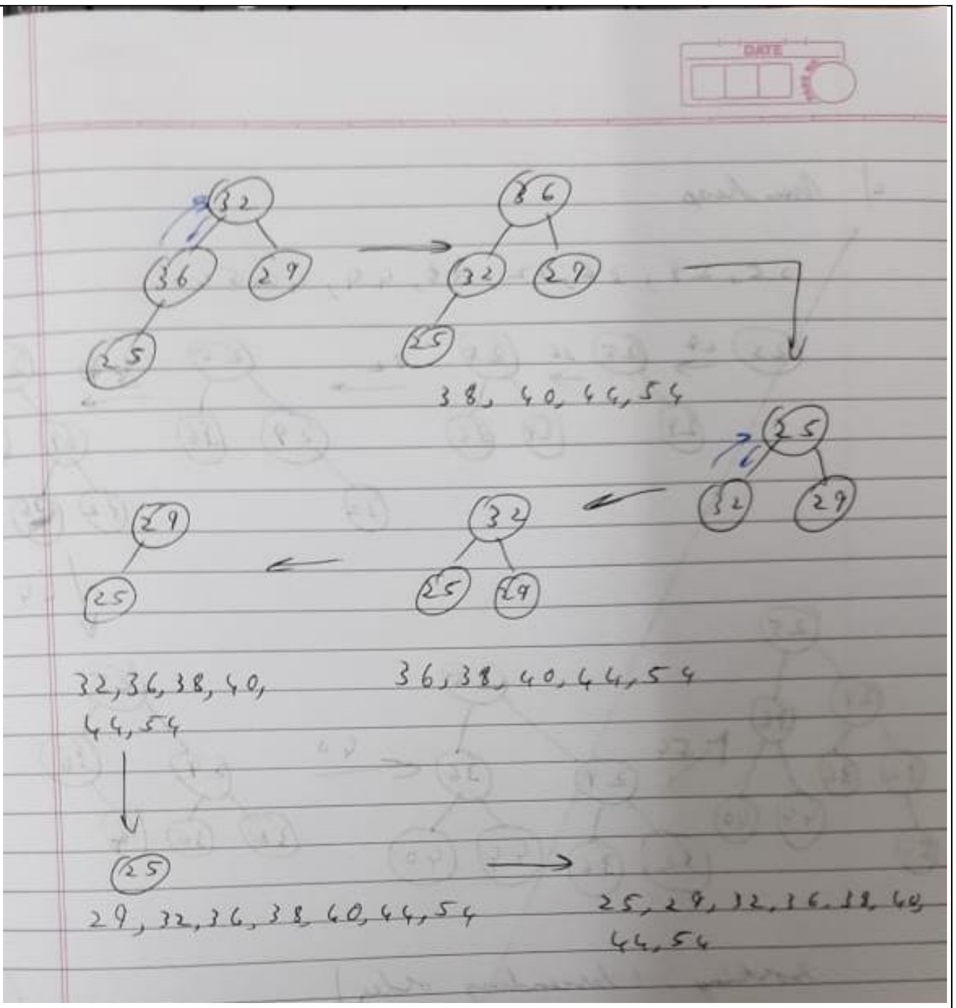
- **Lack of flexibility:** The heap data structure is not very flexible, as it is designed to maintain a specific order of elements. This

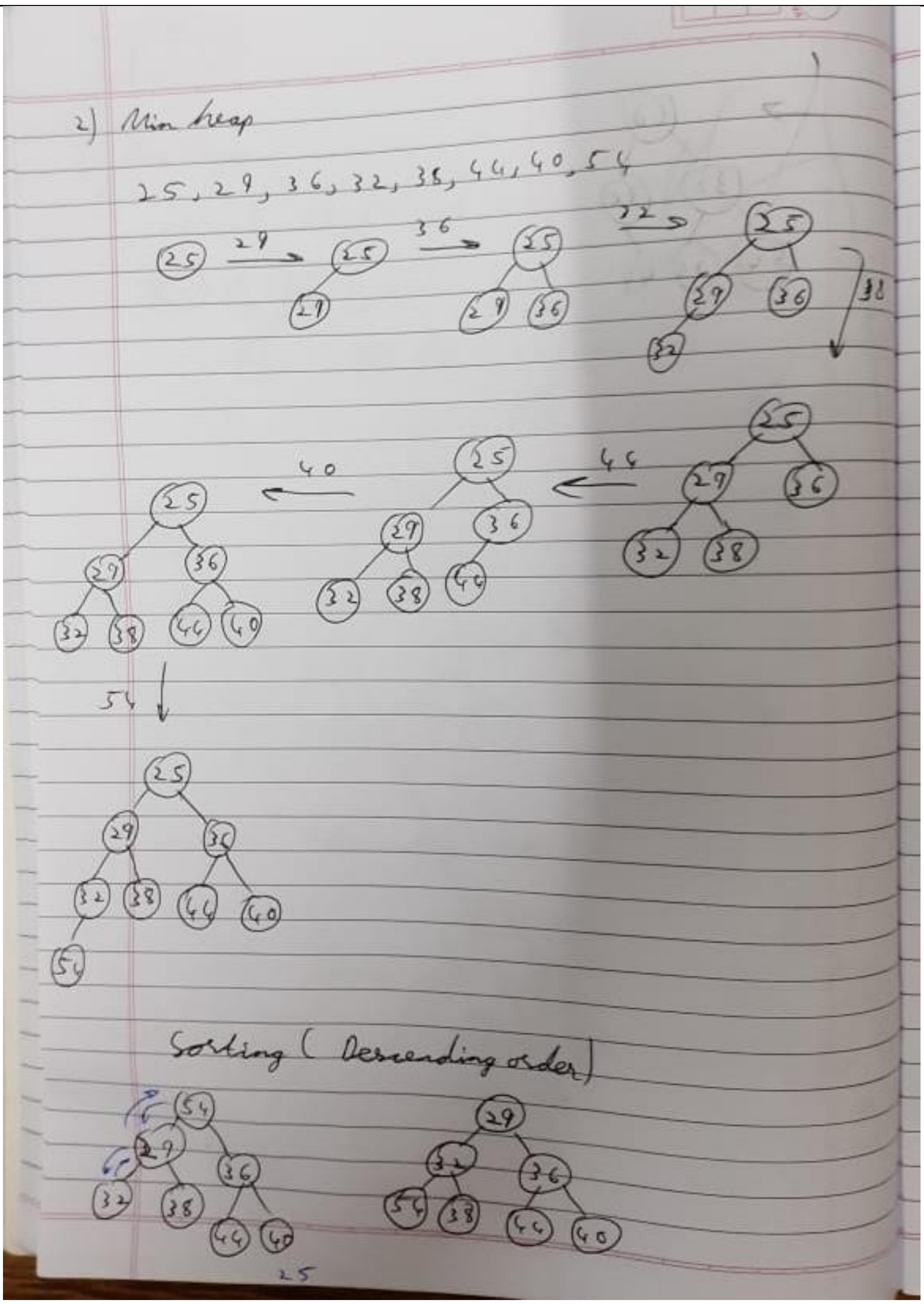
	means that it may not be suitable for some applications that require more flexible data structures. • **Not ideal for searching:** While the heap data structure allows efficient access to the top element, it is not ideal for searching for a specific element in the heap. Searching for an element in a heap requires traversing the entire tree, which has a time complexity of O(n). • **Not a stable data structure:** The heap data structure is not a stable data structure, which means that the relative order of equal elements may not be preserved when the heap is constructed or modified.
Algorithm	**Heap Sort Algorithm** First convert the array into a max heap using **heapify**, this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1. • Rearrange array elements so that they form a Max Heap. • Repeat the following steps until the heap contains only one element: o Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap. o Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array. o Heapify the remaining elements of the heap. • Finally we get sorted array. 1) **Heapify Function**: • heapify is a recursive function that enforces the heap property for a subtree rooted at a given index. • For a given index i, it checks the left and right child nodes to identify the largest (in Max-Heap mode) or smallest (in Min-Heap mode) value among the three nodes (parent and two children). • If the largest or smallest node is not the root, it swaps the values and recursively calls heapify on the affected subtree. 2) **Building the Heap**: • In the heapSort function, the initial for-loop builds the heap from the input array by calling heapify on each non-leaf node in reverse order (starting from n/2 - 1 down to 0). • This process rearranges the array into a Max-Heap or Min-Heap. 3) **Heap Sort Process**: • After the heap is built, the function enters a loop where it repeatedly swaps the root of the heap (the largest or smallest element) with the last element of the unsorted portion of the array.

	• After each swap, it reduces the heap size by one and calls heapify on the root to re-establish the heap property for the remaining elements. 4) **Iterative Sorting**: • With each iteration, the largest (or smallest) value moves to its final sorted position at the end (or beginning) of the array. • This continues until only one element remains in the heap, resulting in a fully sorted array.
Problem Solving	 David Daniels 202330038 25, 29, 36, 32, 38, 44, 40, 59 1) Max heap 29 36 32 38 44 40

54
Sorting : (ascending order)
54
44, 54
40, 44, 54

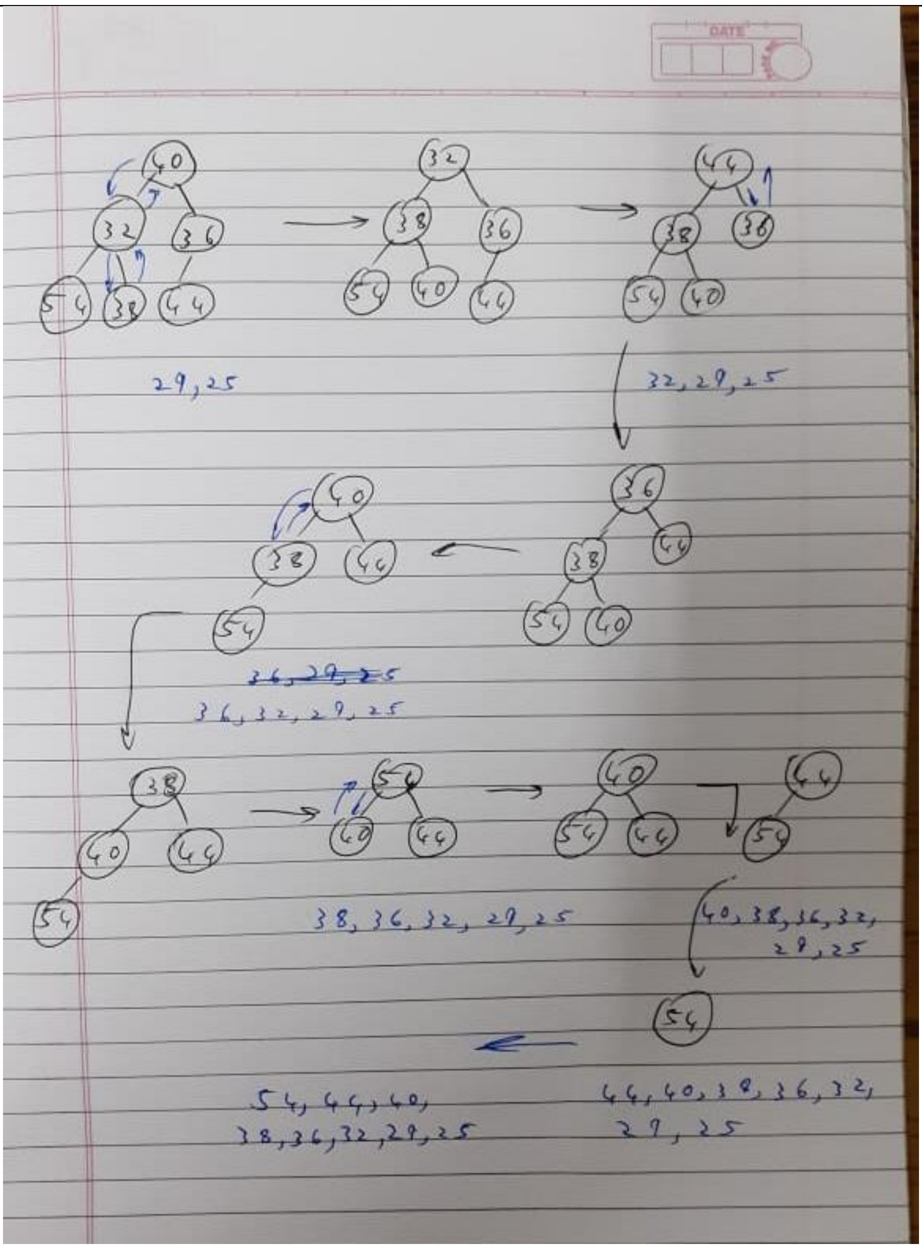
	 32 36 29 25 36 32 29 25 38, 40, 44, 54 25 32 29 32 25 29 29 25 32, 36, 38, 40, 44, 54 36, 38, 40, 44, 54 25 29, 32, 36, 38, 40, 44, 54 25, 29, 32, 36, 38, 40, 44, 54

2) Min heap

25, 29, 36, 32, 38, 44, 40, 54

Sorting (Descending order)

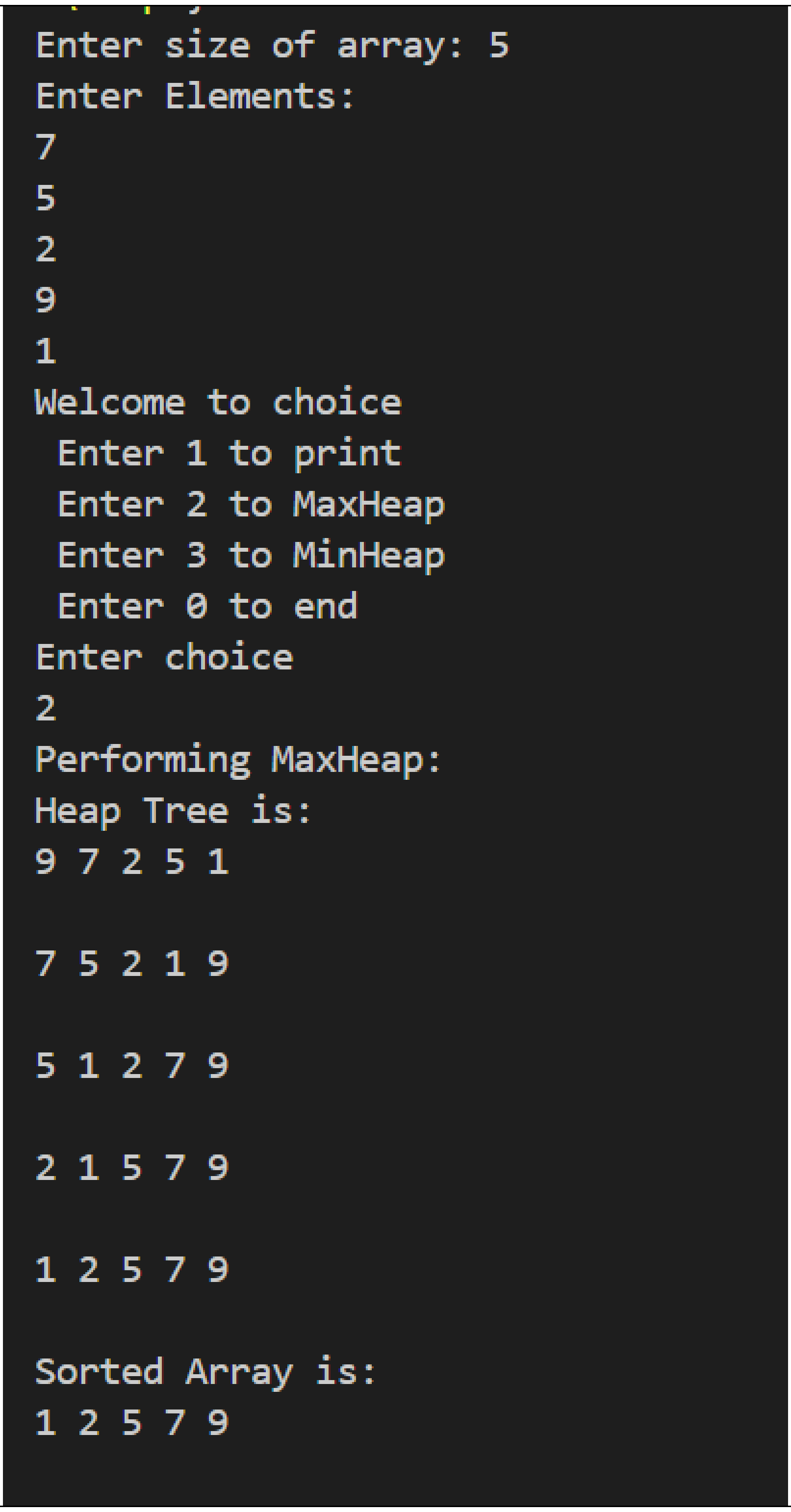


Program(Code)

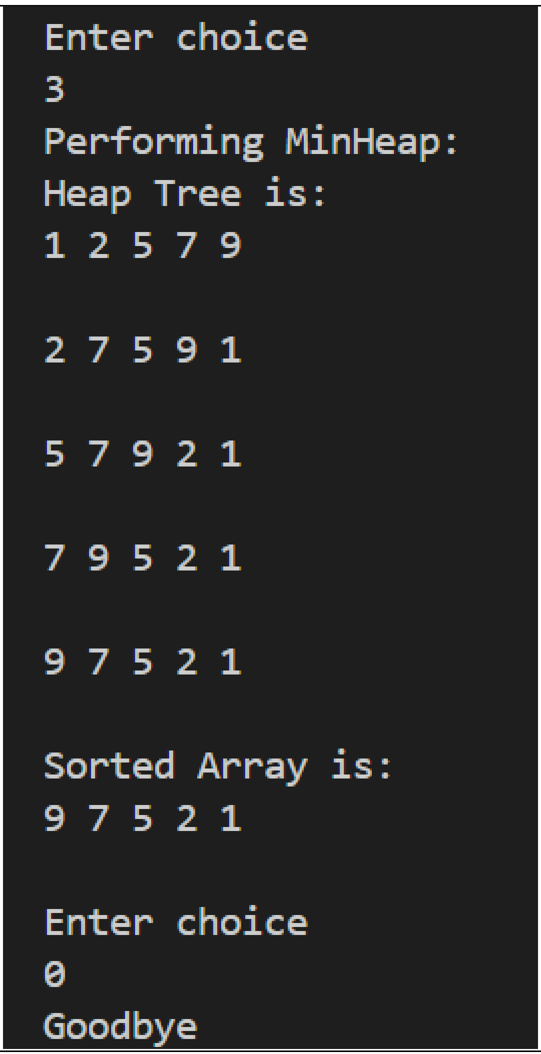
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

void print(int * arr,int size)
{
   for (int i = 0; i < size; i++)
   {
      printf("%d ",arr[i]);
   }
   printf("\n\n");
}
```

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

void heapify(int * arr, int n, int i, int isMinHeap) {
    int largest_smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (isMinHeap) {
        if (left < n && arr[left] < arr[largest_smallest])
            largest_smallest = left;
        if (right < n && arr[right] < arr[largest_smallest])
            largest_smallest = right;
    } else {
        if (left < n && arr[left] > arr[largest_smallest])
            largest_smallest = left;
        if (right < n && arr[right] > arr[largest_smallest])
            largest_smallest = right;
    }

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

void heapSort(int * arr, int n, int isMinHeap) {
    //build heap from array
    for (int i = n / 2 - 1; i >= 0; i--){
        heapify(arr, n, i, isMinHeap);
    }
    printf("Heap Tree is: \n");
    print(arr,n);
    //swap it
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0, isMinHeap);
        print(arr,n);
    }
    printf("Sorted Array is: \n");
    print(arr,n);
}

void choice (int * arr,int size)
{
    int choice=0;
    printf("Welcome to choice \n Enter 1 to print \n Enter 2 to MaxHeap \n Enter 3 to MinHeap \n Enter 0 to end\n");
    int num=0,len=0;
    while (true)
    {
        printf("Enter choice \n");
        scanf("%d",&choice);
        if (choice==0)
        {
            printf("Goodbye");
            break;
        }

        switch (choice)
        {
        case 2:
            printf("Performing MaxHeap: \n");
            heapSort(arr, size, 0);
            break;
        case 3:
            printf("Performing MinHeap: \n");
            heapSort(arr, size, 1);
            break;
        case 1:
            print(arr,size);
            break;
        default:
            printf("Error");
            break;
        }
    }
}

int main(int argc, char const *argv[])
{

    int size=0;
    printf("Enter size of array: ");
    scanf("%d",&size);
    int * arr=(int *)malloc(size*sizeof(int));
    printf("Enter Elements: \n");
    int temp=0;
    for (int i = 0; i < size; i++)
    {
        scanf("%d",&temp);
        arr[i]=temp;
    }
    choice(arr,size);

    free(arr);
```

	return 0;
	}

Output	

```
Enter size of array: 5
Enter Elements:
7
5
2
9
1
Welcome to choice
 Enter 1 to print
 Enter 2 to MaxHeap
 Enter 3 to MinHeap
 Enter 0 to end
Enter choice
2
Performing MaxHeap:
Heap Tree is:
9 7 2 5 1

7 5 2 1 9

5 1 2 7 9

2 1 5 7 9

1 2 5 7 9

Sorted Array is:
1 2 5 7 9
```



```
Enter choice
3
Performing MinHeap:
Heap Tree is:
1 2 5 7 9

2 7 5 9 1

5 7 9 2 1

7 9 5 2 1

9 7 5 2 1

Sorted Array is:
9 7 5 2 1

Enter choice
0
Goodbye
```

Conclusion

HeapSort is an efficient sorting algorithm with a time complexity of O(nlogn), making it well-suited for large data sets. This experiment demonstrated HeapSort using both Min-Heap and Max-Heap structures, allowing the array to be sorted in ascending or descending order. HeapSort's in-place sorting and non-recursive nature contribute to its space efficiency, while its systematic heap construction minimizes the number of swaps, giving it an advantage over certain other sorting methods in specific applications.