NAME	Sujal Dingakar
UID no	2024301005
Experiment No	10

AIM:	**Implement the HeapSort algorithm (Min/Max). The program should accept an ARRAY A as input. The output should be Sorted Array A.**
THEORY:	**Why Do We Use HEAP?** Heaps are used primarily for their efficiency in handling priority-based operations. They are essential for situations where you need to quickly retrieve the maximum or minimum element, such as in priority queues. Heaps are useful for tasks like sorting (Heap Sort), dynamic memory allocation, and implementing algorithms like Dijkstra's shortest path algorithm. Heaps allow for efficient extraction of the highest or lowest priority element, making them suitable for scheduling tasks, implementing priority queues, and other scenarios where you need to manage a dynamic set of elements with quick access to the extreme values. **What is Heap and What Are Its Types?** A **heap** is a specialized tree-based data structure that satisfies the **heap property**: o In a **max heap**, the key of each parent node is greater than or equal to the keys of its children, and the maximum element is at the root. o In a **min heap**, the key of each parent node is less than or equal to the keys of its children, and the minimum element is at the root. Heaps are often implemented as binary heaps, but other types include **Fibonacci heaps** and **Binomial heaps**. **Advantages of Hashing Over Other Data Structures** • **Heaps offer several advantages over other data structures:** • **Efficient priority retrieval:** Heaps allow the retrieval of the maximum (or minimum) element in constant time, i.e., O(1)O(1)O(1). • **Efficient insertion and deletion:** Insertion and deletion operations in heaps are more efficient than in other data structures like arrays and linked lists, with time complexity of O(logn)O(\log n)O(logn). • **Balanced tree structure:** The heap's structure is always balanced, ensuring optimal time complexity for its operations, unlike binary search trees (BST) that can degrade into linked lists.

Heap sort: Heap sort is an in-place sorting algorithm with O(nlogn)O(n \log n)O(nlogn) time complexity, offering better performance than algorithms like bubble sort and selection sort.
Heaps are particularly useful in scenarios where you need to frequently access or modify the highest or lowest value in a collection, such as in scheduling, event-driven simulations, and priority queues.

Operations That Can Be Performed on Heap?
Heaps support several fundamental operations:
Insert: Add an element to the heap, ensuring the heap property is maintained. This operation is done in O(log⁡n)O(\log n)O(logn) time.
Delete: Remove the root element (max or min) from the heap. This is also done in O(logn)O(\log n)O(logn) time.
Peek: Retrieve the root element without removing it. This operation is O(1)O(1)O(1) because the root is always the maximum (or minimum).
Heapify: Convert a collection of elements into a valid heap, which is done in O(n)O(n)O(n) time.
Extract: Remove the root element (max or min), then restore the heap property by "heapifying" the tree, which takes O(log⁡n)O(\log n)O(logn).

Applications of Heap?

- **Answer: Heaps are used in a variety of applications:**
    - **Priority Queue:** Heaps are used to implement priority queues, where elements are retrieved based on priority rather than insertion order.
    - **Heap Sort:** A sorting algorithm that uses a binary heap to sort elements in O(nlogn)O(n \log n)O(nlogn) time.
    - **Dijkstra's Algorithm**: Used for finding the shortest path in a graph, heaps can efficiently manage the priority of nodes to be processed.
    - **Dynamic Memory Allocation:** Heaps can be used for managing free and allocated memory in an efficient manner.
    - **Kth Largest/Smallest Element:** Heaps are helpful when you need to find the k-th largest or smallest element in a dataset.
    - **Merging Multiple Sorted Files**: In external sorting, heaps are used to merge multiple sorted files by efficiently extracting the minimum (or maximum) element from each file.

What Is a Fibonacci Heap and When to Use It?
A Fibonacci heap is an advanced heap structure that supports operations such as insert, delete, and decrease key with improved amortized time complexities compared to binary heaps. It is particularly useful in algorithms like Dijkstra's and Prim's where multiple decrease-key operations are needed. The Fibonacci heap provides

	O(1)O(1)O(1) amortized time for insertion and O(logn)O(\log n)O(logn) amortized time for extraction, making it very efficient for specific types of problems. **Is Heap useful in REAL LIFE SCENARIO ?** Heaps are used in real-life applications like priority queues, task scheduling, and Dijkstra's algorithm for efficient shortest path finding. They enable quick access to the maximum or minimum element, making them ideal for real-time systems like stock monitoring and event-driven simulations. Heaps are also used in merging sorted files, dynamic memory allocation, and finding the k-th largest or smallest element in data streams. Their ability to maintain order efficiently makes them essential in managing data in real-time and large-scale systems. **What is Heapify ?** • Heapify is a process to rearrange elements to maintain heap property • It ensures that parent nodes maintain correct relationship with children Can be used for both min-heap and max-heap. **How to Perform Heapify:** • First create the Complete binary tree • Start from bottom (new inserted element) • Compare with parent o Swap if heap property is violated • Continue until root or heap property is satisfied • Time complexity: O(log n) **Extract Min/Max** : The extract min/max concept refers to an operation in heap data structures where the root element (which is either the minimum or maximum value, depending on the type of heap) is removed and returned. This operation is essential in maintaining priority queues and performing algorithms like heap sort. **Explanation of Extract Min/Max:** • **Extract Max**: Used in a max heap, where the maximum value is always at the root. • **Extract Min**: Used in a min heap, where the minimum value is always at the root. **How Extract Min/Max Works:** **Remove the Root**: • The element at the root (max or min) is removed because it is the highest priority element.

	Replace the Root: • Replace the root with the last element of the heap (the rightmost element in the last level). **Heapify (Down-Heapify):** • Perform a heapify operation starting from the root to maintain the heap property. • In a max heap, ensure the root is greater than its children by swapping with the larger child if necessary. • In a min heap, ensure the root is smaller than its children by swapping with the smaller child if necessary.
ALGORITHM**:**	**Define Helper Functions:** • print(arr, n): Print elements of an array from index 1 to n. • swap(a, b): Swap the values at two given memory addresses. • heapifyMax(arr, n, i): Maintain the max-heap property for a subtree rooted at index i. o Let largest be i, left be 2*i, and right be 2*i + 1. o If left is within the array and arr[left] > arr[largest], set largest to left. o If right is within the array and arr[right] > arr[largest], set largest to right. o If largest is not i, swap arr[i] with arr[largest] and recursively call heapifyMax on largest. **Build Max Heap (buildMaxHeap):** • For each node starting from n/2 down to 1, call heapifyMax to ensure the max-heap property. • This converts the array into a max-heap structure. **Heap Sort (heapSortMax):** • For each element from the last index n down to 1: o Print the current iteration and the heap state, excluding the sorted part. o Swap the root element (maximum) with the last unsorted element. o Reduce the unsorted array size by 1. o Call heapifyMax on the reduced heap to maintain the max-heap property. o Print the state of the partially sorted heap. **Main Program Execution:** • Prompt the user for the array size and elements. • Build a max heap from the array. • Perform heap sort on the max heap. • Print the final sorted array.
PROBLEM SOLVING:	

Subject - Data Structure.

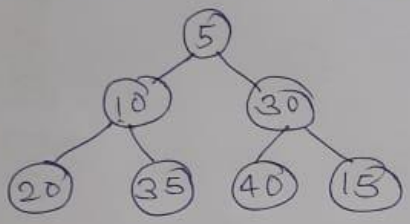
Topic - Implement the heapSort algorithm.

Name - Sujal Sandeep Dinganhar.

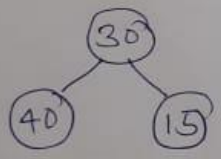
Date - 09-11-2024.

Input → 5, 10, 30, 20, 35, 40, 15

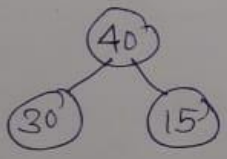
Step 1 = First create a complete binary tree -



Step 2 = We will calculate a last - non-leaf (Internal elements) $n/2-1 = 4/2-1 = 3-1 = 2$



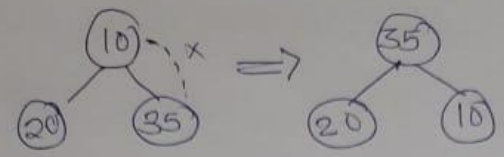
Step 3 = Check the heap property

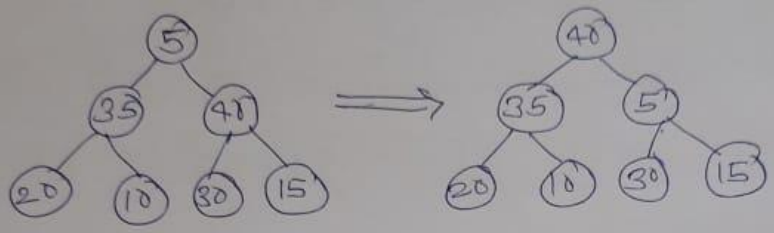


array - 5 10 30 20 35 40 15

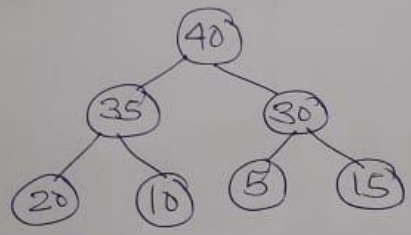
New array (updated) = 5 10 40 20 35 30 15

Step4 = Then check for next non-leaf element





go towards end →



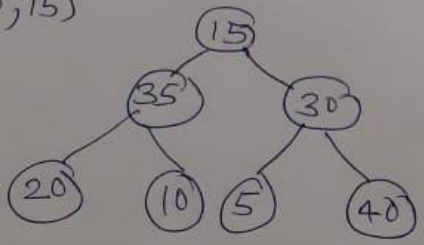
array => | 40 | 35 | 30 | 20 | 10 | 5 | 15 |

Max-heap

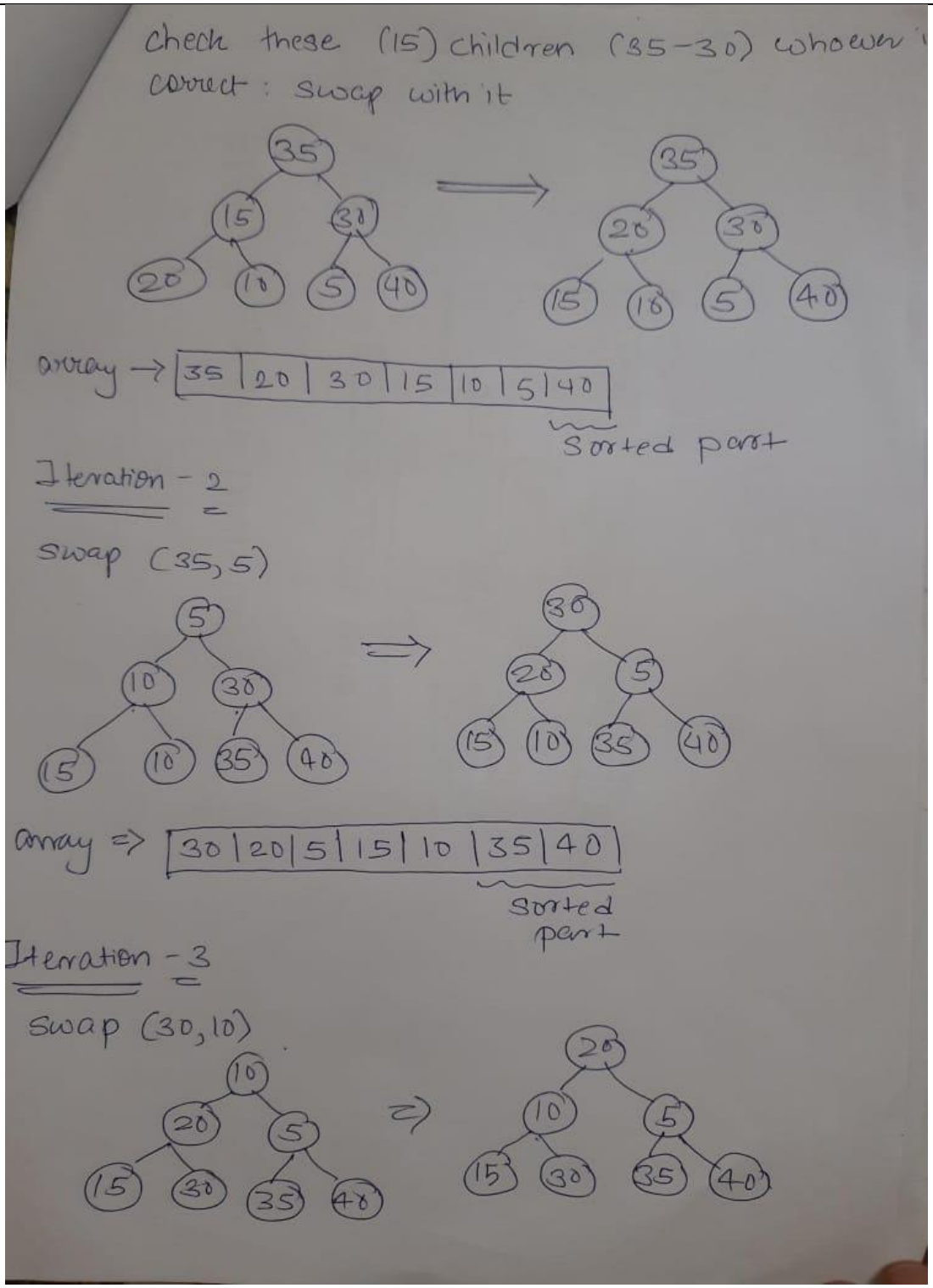
- Iteration 1:-

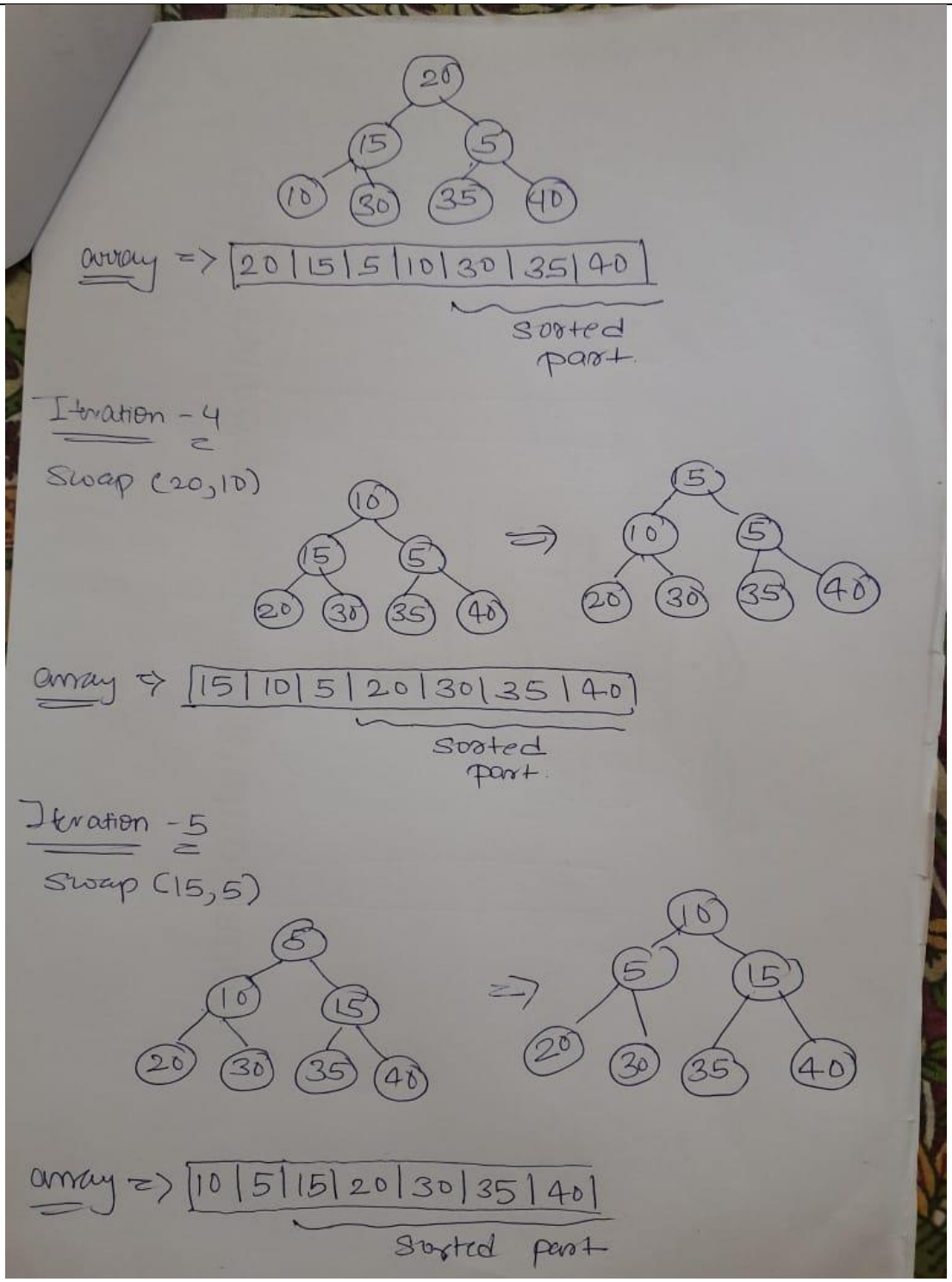
extract max - now swaps root with last element

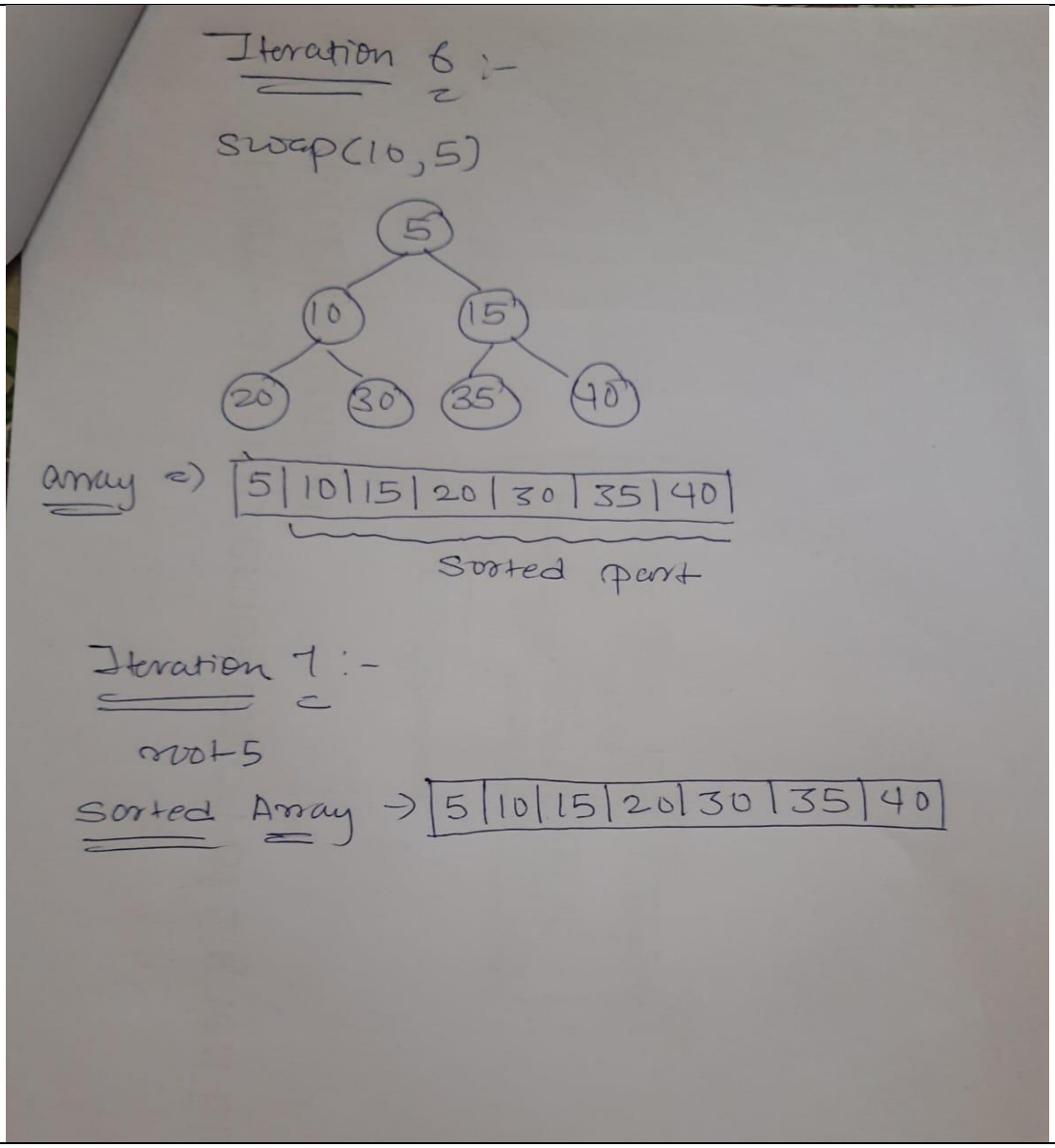
swap (40, 15)



Now again perform heapify -


check these (15) children (35-30) whoever
correct : swap with it
35
15
30
20
10
5
40
35
20
30
15
10
5
40
array → 35 | 20 | 30 | 15 | 10 | 5 | 40
Sorted part
Iteration - 2
swap (35, 5)
5
10
30
15
10
35
40
30
20
5
15
10
35
40
array => 30 | 20 | 5 | 15 | 10 | 35 | 40
sorted part
Iteration - 3
swap (30, 10)
10
20
5
15
30
35
40
20
10
5
15
30
35
40


20
15
5
10
30
35
40
array => 20 | 15 | 5 | 10 | 30 | 35 | 40
sorted part
Iteration - 4
Swap (20, 10)
10
15
5
20
30
35
40
5
10
5
20
30
35
40
array => 15 | 10 | 5 | 20 | 30 | 35 | 40
sorted part
Iteration - 5
Swap (15, 5)
5
10
15
20
30
35
40
10
5
15
20
30
35
40
array => 10 | 5 | 15 | 20 | 30 | 35 | 40
sorted part



PROGRAM:

```
#include <stdio.h>

void print(int arr[], int n) {
    for (int i = 1; i <= n; i++) {
        printf("%d ", arr[i]);
    }
}

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

void heapifyMax(int arr[], int n, int i) {
```

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

    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]);
        heapifyMax(arr, n, largest);
    }
}

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

void heapSortMax(int arr[], int n) {
    int var = n;
    int it = 1;

    for (int i = n; i > 0; i--) {
        printf("\n\n\nIteration: %d", it);
        printf("\nPrinting the heap elements (ignoring sorted part)\n");
        print(arr, var);
        printf("\nRoot %d swapped with: %d\n", arr[1], arr[i]);
        swap(&arr[1], &arr[i]);
        var--;
        heapifyMax(arr, var, 1);
        printf("Partially sorted heap (after Max-Heapifying unsorted part)\n");
        print(arr, n);
        it++;
    }
}

int main() {
    int size = 0;
    printf("Enter the array size: ");
```

	scanf("%d", &size);
	int array[size];
	
	for (int i = 1; i <= size; i++) {
	printf("\nEnter element %d: ", i);
	scanf("%d", &array[i]);
	}
	
	printf("\nThe unordered heap is:\n");
	print(array, size);
	
	printf("\nBuilding a max heap:\n");
	buildMaxHeap(array, size);
	print(array, size);
	
	printf("\n\nLet us perform a heapsort:");
	heapSortMax(array, size);
	
	printf("\nSorted Heap after heapsort: \n");
	print(array, size);
	
	return 0;
	}

RESULT :-	

```
Enter the array size: 7

Enter element 1: 5

Enter element 2: 10

Enter element 3: 30

Enter element 4: 20

Enter element 5: 35

Enter element 6: 40

Enter element 7: 15

The unordered heap is:
5 10 30 20 35 40 15
Building a max heap:
40 35 30 20 10 5 15
```

```
Let us perform a heapsort:


Iteration: 1
Printing the heap elements (ignoring sorted part)
40 35 30 20 10 5 15
Root 40 swapped with: 15
Partially sorted heap (after Max-Heapifying unsorted part)
35 20 30 15 10 5 40


Iteration: 2
Printing the heap elements (ignoring sorted part)
35 20 30 15 10 5
Root 35 swapped with: 5
Partially sorted heap (after Max-Heapifying unsorted part)
30 20 5 15 10 35 40


Iteration: 3
Printing the heap elements (ignoring sorted part)
30 20 5 15 10
Root 30 swapped with: 10
Partially sorted heap (after Max-Heapifying unsorted part)
20 15 5 10 30 35 40
```

```
Iteration: 4
Printing the heap elements (ignoring sorted part)
20 15 5 10
Root 20 swapped with: 10
Partially sorted heap (after Max-Heapifying unsorted part)
15 10 5 20 30 35 40


Iteration: 5
Printing the heap elements (ignoring sorted part)
15 10 5
Root 15 swapped with: 5
Partially sorted heap (after Max-Heapifying unsorted part)
10 5 15 20 30 35 40


Iteration: 6
Printing the heap elements (ignoring sorted part)
10 5
Root 10 swapped with: 5
Partially sorted heap (after Max-Heapifying unsorted part)
5 10 15 20 30 35 40


Iteration: 7
Printing the heap elements (ignoring sorted part)
5
Root 5 swapped with: 5
Partially sorted heap (after Max-Heapifying unsorted part)
5 10 15 20 30 35 40
Sorted Heap after heapsort:
5 10 15 20 30 35 40
```

CONCLUSION :-	The heap property ensures that a binary tree is a complete binary tree where each parent node follows a specific order: in a max heap, parents are greater than or equal to their children, and in a min heap, parents are smaller. Heapify maintains this property by adjusting the tree structure, while extracting max/min removes the root and restores the heap. Heap sort uses the heap property to efficiently sort an array by repeatedly extracting the max/min element. Overall, heaps offer an efficient way to manage priorities and perform sorting.