NAME: ODEYEMI TEMITOPE BENJAMIN

CLASS: MPhil/PhD

24000058

HEAP SORT

Heap sort is a comparison-based sorting algorithm that sort numbers in order by using the structure if a heap, tree-like structure in which the maximum number have the highest priority.

Imagine the pyramid of books where the tallest book stays at the **TOP**. The process of arranging or heap sorting is called **HEAPIFYING**.

Heap Sort Paradigm

Paradigm of heap sort is based on divide and conquer approach and uses the heap data structure (a specialized binary tree). The only difference between heap sort and other divide and conquer algorithms is that heap sort does not require recursion; instead it uses iterative approach with a single array as the heap.

1. **Divide and conquer:**

- Heap sort divides the array into 2 logical parts: the unsorted portion and the sorted portion
- By repeatedly "conquering" the maximum or minimum element from the heap and moving it to the sorted portion, it achieves a fully sorted array.

2. **Heap-based paradigm:**

- The sort uses binary heap, where the max-heap is identify as the parent node which must be greater or equal to its children (for sorting in ascending order). While, the Min-heap is the parent node which is smaller than or equal to its children (for sorting in descending order).

Steps involve in Heap Sort

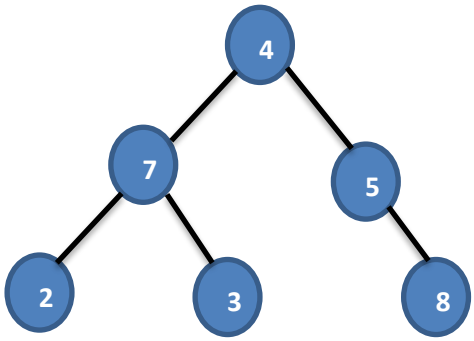
1. **Build a heap:** The input array is transform into max-heap (or min-heap) to ensure that the largest (or smallest) element is at the root of the heap.
2. **Extract elements:** The root element (largest or smallest) is swapped with the last elements in the heap and removed, thereby reducing the size of the heap and the heap property is restored by a process called "heapifying".
3. **Repeat:** Extraction and heapifying process is repeated until the heap is empty and the array is completely sorted.

Example

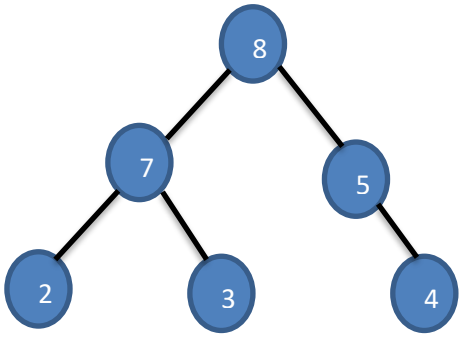
Input: 4, 7, 5, 2, 3, 8

4	**7**	**5**	**2**	**3**	**8**

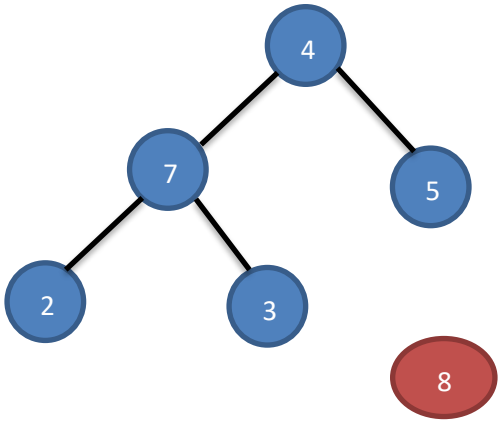
STEP 1: Build-up a heap



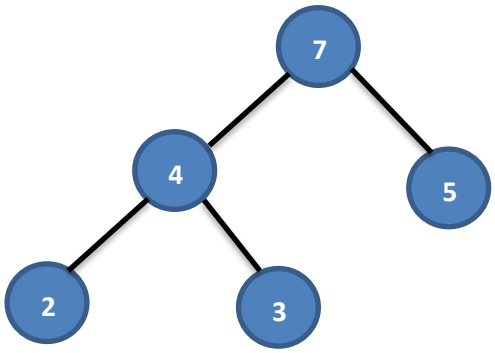
STEP 2: Swap 8 and 4



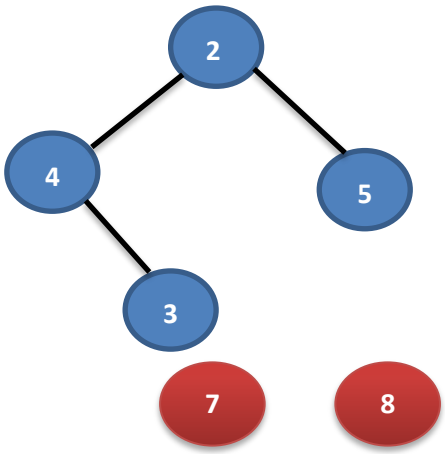
STEP 3: Remove 8 from the heap, replace the root with node 4



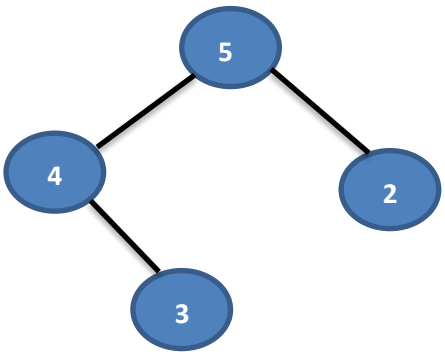
STEP 4: Swap 7 and 4



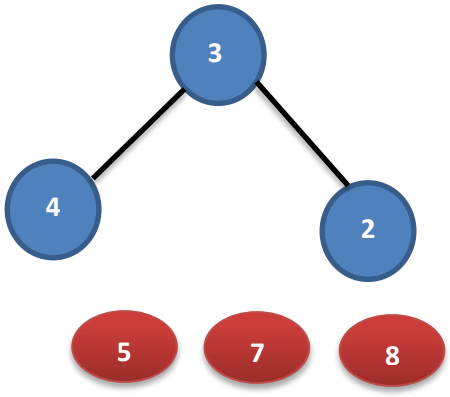
STEP 5: Drop 7 from the heap and replace with 2



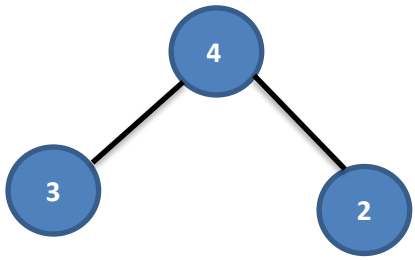
STEP 6: Swap 5 and 2



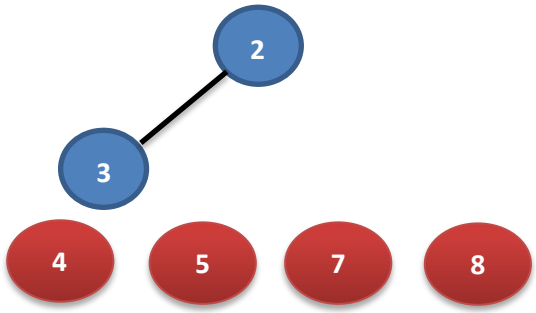
STEP 7: Remove 5 from the heap and replace with 3



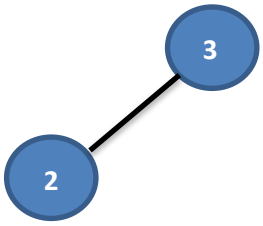
STEP 8: Swap 4 with 3



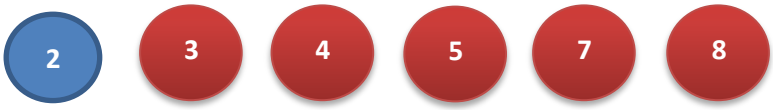
STEP 9: Remove 4 from the heap and replace with 2



STEP 10: Swap 3 and 2



STEP 11: Remove 3 from the heap and replace with 2



Heap Algorithm complexity

Best time complexity: Logarithm, O (n log n)

Best time complexity: Logarithm, O (n log n)

Space complexity: Constant, O (1)

Advantages of Heap Sort

1. **Consistent Time Complexity:** O (n log n) for best, average, and worst cases.
2. **In-Place Sorting:** Requires only a constant or no amount of additional memory (O (1), **space complexity**).
3. **Not Recursive:** Can be implemented iteratively, avoiding the risk of stack overflow in recursive algorithms.
4. Efficient for Large Data Sets.

Disadvantages of Heap Sort

1. Does not maintain the relative order of equal elements.
2. More complex to implement compared to simpler sorting algorithms like insertion sort or selection sort.
3. Access patterns can lead to less efficient use of the CPU cache compared to algorithms like quicksort.
4. Performance does not improve for partially sorted arrays.

Applications of Heap Sort

1. Efficiently handles large arrays or lists due to its O (n log n) time complexity.
2. Used to implement priority queues where elements are processed based on priority.
3. Suitable for real-time systems that require guaranteed time complexity.
4. Useful in external sorting algorithms where data is too large to fit into memory.
5. Applied in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree, which utilize heap structures.
6. Used in discrete event simulation systems where the next event needs to be processed in priority order.

Heap sort Pseudo code

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Heap sort code in Python (Design to accept input from keyboard)

```python
# Python program for implementation of heap Sort

def heapify(arr, n, i):
    # Initialize largest as root
    largest = i

    # left index = 2*i + 1
    l = 2 * i + 1

    # right index = 2*i + 2
    r = 2 * i + 2

    # If left child is larger than root
    if l < n and arr[l] > arr[largest]:
        largest = l

    # If right child is larger than largest so far
    if r < n and arr[r] > arr[largest]:
        largest = r

    # If largest is not root
```

```
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap

        # Recursively heapify the affected sub-tree
        heapify(arr, n, largest)

# Main function to do heap sort
def heapSort(arr):
    n = len(arr)

    # Build heap (rearrange array)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract an element from heap
    for i in range(n - 1, 0, -1):
        # Move root to end
        arr[0], arr[i] = arr[i], arr[0]

        # Call max heapify on the reduced heap
        heapify(arr, i, 0)

def printArray(arr):
    for i in arr:
        print(i, end=", ")
    print()

# Input code and display
try:
    input_data = input("Enter a list of numbers (integers or decimals) separated
by spaces: ")
    arr = list(map(float, input_data.split()))  # Changed to `float` to handle
decimals

    print("Input array is ")
    printArray(arr)
    heapSort(arr)
    print("Sorted array is ")
    printArray(arr)
except ValueError:
    print("Please enter valid numbers (integers or decimals).")
```

BUCKET SORT

Bucket sort is a non-comparison based algorithm named distribution sorting, for organizing numbers by first grouping them into smaller manageable units called buckets based on their size and then sorts the numbers within each bucket individually.

Paradigm of Bucket sort

The paradigm is based on distribution sorting in combination with divide and conquers.

1. **Distribution sorting:** The input elements are distributed into different buckets according to their range.
2. **Divide and conquer:** The array is divided into smaller units called buckets, and then the buckets are sorted individually using simpler algorithms such as insertion sort.
3. **Combination:** After sorting individual buckets, the data in all the buckets are combined together to form final sorted array.

Steps involve in Bucket sort

1. **Bucket creation:** Determine the length of the input data and create several empty buckets of equal interval.
2. **Distribution:** Place each element of the input array into its appropriate bucket based on its value.
3. **Sorting Buckets:** Sort the content of each bucket individually using simpler sorting algorithm such as insertion or quick sort.
4. **Merging:** Combine sorted elements in all buckets together to form a single array.

Advantages of Bucket sort

1. It's more efficient for uniformly distributed data.
2. It is simple to use and understand.
3. Sorting is done parallel method.
4. It minimizes comparison by leveraging distribution.

Disadvantages of Bucket sort

1. It requires additional memory to store the buckets
2. If the uniformity of data inside each bucket is not the same, there may be increase in time complexity.
3. Sorting within each bucket require additional computational cost.
4. It is more efficient in sorting floating-point numbers.

Application area of Bucket sort

1. Sorting sensor readings in real-time control systems.
2. Sorting financial transaction amounts for analysis and fraud detection.
3. Sorting pixels based on brightness for image rendering.
4. Sorting exam scores for a class of students.

Example

Step 1: Create buckets

Given input:

0.31	**0.52**	**0.84**	**0.72**	**0.45**	**0.92**

Bucket for this **input** are;

B1: 0.0-0.2, **B2:** 0.2-0.4, **B3:** 0.4-0.6, **B4:** 0.6-0.8, **B5:** 0.8-1.0

Step 2: Distribute numbers into buckets

0.31 -------- **B2**

0.52 -------- **B3**

0.84 -------- **B5**

0.72 -------- **B4**

0.45 -------- **B3**

0.92 -------- **B5**

Bucket after distribution:

B1: [ ], **B2:** [0.31], **B3:** [0.52, 0.45], **B4:** [0.72], **B5:** [0.84, 0.92]

Step 3: Sort each bucket using other algorithm such as insertion or quick sort etc.

B1: [ ], **B2:** [0.31], **B3:** [0.45, 0.52], **B4:** [0.72], **B5:** [0.84, 0.92]

Step 4: Combine the buckets

0.31	**0.45**	**0.52**	**0.72**	**0.84**	**0.92**

Bucket sort complexity

Best or Average case: Linear, O (n + k)

Worst case: Quadratic, O (n^2)

Space complexity: Linear, O (n + k)

Where, **n** is the number of element and **k** is the number of bucket

Bucket sort Pseudo code

Bucket sort code in Python (Design to accept input from keyboard)

```
def insertion_sort(bucket):
    for i in range(1, len(bucket)):
        key = bucket[i]
        j = i - 1
        while j >= 0 and bucket[j] > key:
            bucket[j + 1] = bucket[j]
            j -= 1
        bucket[j + 1] = key

def bucket_sort(arr):
    n = len(arr)
    if n == 0:
        return

    # Find the maximum and minimum values
```

```
    min_val = min(arr)
    max_val = max(arr)

    # Normalize numbers to the range [0, 1]
    range_val = max_val - min_val
    if range_val == 0:  # All elements are the same
        return

    buckets = [[] for _ in range(n)]

    # Put array elements in different buckets
    for num in arr:
        normalized_index = int((num - min_val) / range_val * (n - 1))
        buckets[normalized_index].append(num)

    # Sort individual buckets using insertion sort
    for bucket in buckets:
        insertion_sort(bucket)

    # Merge all buckets into arr[]
    index = 0
    for bucket in buckets:
        for num in bucket:
            arr[index] = num
            index += 1

# Input numbers from the keyboard
try:
    input_data = input("Enter a list of numbers separated by spaces (e.g., 0.23 2
-1.5): ")
    arr = list(map(float, input_data.split()))

    print("Input array is:")
    print(arr)

    bucket_sort(arr)

    print("Sorted array is:")
    print(" ".join(map(str, arr)))
except ValueError:
    print("Please enter valid numbers.")
```