CS 514

Assignment 3 – Heapsort

report3.txt

Q1. Implement the heapSort algorithm. Given an input array, the heapSort algorithm acts as follows:

- **Builds a max-heap from the array [1..] (instead of the min-heap provided in the exploration).**
- **Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.**
- **Discard this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.**
- **Repeat this discarding process until only one node (the smallest element) remains, and therefore is in the correct place in the array.**

Answer-

Please find the implementation of heap_sort which sorts numbers in increasing order in a given array.

```
#
# Max-Heapify function to maintain the max-heap property.
def max_heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)

# Build a max-heap from the array.
def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)


# Heap Sort function.
def heap_sort(arr):
    n = len(arr)
    # Build a max-heap from the array.
    build_max_heap(arr)
```

```
    # Starting with the root (the maximum element), the algorithm places the
maximum element into the correct place in the array by swapping it with the
element in the last position in the array.
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
       # Condition 3: Discard this last node (knowing that it is in its
correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the
new (possibly incorrectly-placed) root.
        max_heapify(arr, i, 0)
    return arr

if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]

    # Call Heap Sort and print the sorted array.
    sorted_arr = heap_sort(arr)
    print(sorted_arr)
```

2. Implement a minPriorityQueue using a min-heap. Given a heap, the minPriorityQueue acts as follows:

- **insert() – insert an element with a specified priority value**
- **first() – return the element with the lowest/minimum priority value (the "first" element in the priority queue)**
- **remove_first() – remove (and return) the element with the lowest/minimum priority value**

Solution -

```
class minPriorityQueue:
    def __init__(self):
        # Initialize the minPriorityQueue with an empty heap.
        self.heap = []

    def insert(self, value):
        # Insert an element with a specified priority value into the priority
queue.
        self.heap.append(value)
        self.percolate_up(len(self.heap) - 1)  # Ensure the heap property is
maintained.

    def first(self):
        # Return the element with the lowest/minimum priority value, i.e.,
the root of the heap.
        if len(self.heap) == 0:
            raise IndexError("Priority queue is empty")
        return self.heap[0]

    def remove_first(self):
        # Remove and return the element with the lowest/minimum priority
value.
        if len(self.heap) == 0:
            raise IndexError("Priority queue is empty")
        if len(self.heap) == 1:
```

```
            return self.heap.pop()

        min_val = self.heap[0]  # The minimum value to be removed.
        last = self.heap.pop()  # Get the last element in the heap.
        self.heap[0] = last  # Replace the root with the last element.
        self.percolate_down(0)  # Ensure the heap property is maintained.
        return min_val

    def percolate_up(self, i):
        # Restore the heap property by moving an element up the heap.
        while i > 0:
            parent = (i - 1) // 2
            if self.heap[i] < self.heap[parent]:
                self.heap[i], self.heap[parent] = self.heap[parent],
self.heap[i]
                i = parent
            else:
                break

    def percolate_down(self, i):
        # Restore the heap property by moving an element down the heap.
        while True:
            smallest = i
            left = 2 * i + 1
            right = 2 * i + 2

            if (left < len(self.heap) and self.heap[left] <
self.heap[smallest]):
                smallest = left

            if (right < len(self.heap) and self.heap[right] <
self.heap[smallest]):
                smallest = right

            if smallest != i:
                self.heap[i], self.heap[smallest] = self.heap[smallest],
self.heap[i]
                i = smallest
            else:
                break

if __name__ == '__main__':
    pq = minPriorityQueue()
    pq.insert(5)
    pq.insert(3)
    pq.insert(7)
    pq.insert(1)
    pq.insert(0)

    print(pq.first())          # Should print the element with the lowest
priority (0).
    print(pq.remove_first())   # Should remove and print the lowest priority
element (0).
    print(pq.first())          # Should print the new element with the lowest
priority (1).
```

3. Argue the correctness of heapsort using the following loop invariant: At the start of each iteration of the for loop, the subarray A[1..i] is a max-heap containing the i smallest elements of A[1..n], and the subarray A[i+1..n]contains the n - i largest elements of A[1..n], sorted.

Note: Here you would assume BUILD-MAX-HEAP and MAX-HEAPIFY are correct as well.

Solution –

Invariant of the loop is - At the start of each iteration of the for loop, the subarray A[1..i] is a max-heap containing the i smallest elements of A[1..n], and the subarray A[i+1..n]contains the n - i largest elements of A[1..n], sorted.

Proof of correctness –

Build-Max-Heap: The loop invariant holds after the build_max_heap function is executed. This is because, after building the max-heap, the largest element (maximum) is at the root of the heap, and the rest of the elements in the heap maintain the max-heap property. This satisfies the loop invariant for i = 0 because A[1..0] is an empty subarray, and A[1..n] contains the n - 0 largest elements, which is the entire array, sorted.

Initialization:
When we start the iteration, we have i = length of the array(n-1). The subarray A[1..n] has n smallest elements and is a max-heap as the build_max_heap function was executed prior to the loop. Hence we have an empty subarray A[n+1..n] which has 0 largest elements of A in sorted order. This satisfies the loop invariant for the initialization condition.

Maintenance:
We assume that the loop invariant holds at the start of the i-th iteration. This means that A[1..i] is a max-heap containing the i smallest elements of A[1..n], and A[i+1..n] contains the n - i largest elements, sorted.

A[1] is the largest element within this subarray A[1..i] as it is a max-heap. As the n - i largest elements are presumed to be at the end of the input array, we can say that A[1] will be the (n - (i - 1))th largest element. Next, we perform a swap operation between A[1] and A[i]. This rearranges A[i..n] to have the (n - i + 1) largest elements of the array, while A[1..i - 1] holds the i - 1 smallest elements.

Now we execute the MAX-HEAPIFY function on A, starting at position 1. Since A[1..i] was previously a max-heap and only the elements at positions 1 and i were exchanged, the left and right subtrees of node 1, up to node i - 1, are assured to remain as max-heaps. The execution of MAX-HEAPIFY will correctly place the element currently residing at node 1 into its appropriate position while restoring the max-heap property, ensuring that A[1..i - 1] remains a max-heap. This ends the current iteration, and as we can see that the loop invariant holds - the subarray A[1..i] is a max-heap containing the i smallest elements of A[1..n], and the subarray A[i+1..n]contains the n - i largest elements of A[1..n] in sorted order.

Termination:
After the last iteration, the loop invariant specifies that the subarray A[2..n] contains the (n - 1) largest elements of A[1..n] in sorted order. As A[1] is the n-th largest element, the entire array is sorted as predicted. Thus, the loop variant holds in termination case as well.