Assignment 2

Qiming Lyu

2025-09-28

1. From problem 6 in Homework 1, we know that the number of internal nodes of a binary heap of size n is $\left\lfloor \frac{n}{2} \right\rfloor$, and the number of leaf nodes is $\left\lceil \frac{n}{2} \right\rceil$. Therefore we conclude that:

   - The number of leaves in a heap of size n is $\left\lceil \frac{n}{2} \right\rceil$.
   - If we remove all leaves from a heap of size n, the remaining heap has size $\left\lfloor \frac{n}{2} \right\rfloor$.

   We remove all leaves once. We get a heap of size $\left\lfloor \frac{n}{2} \right\rfloor$. The leave nodes of the new heap are the nodes at height 1 of the original heap. **The number of them is $\left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2} \right\rceil$.**

2. After removing leaves h times, the size becomes

   $$n^{(h)} = \underbrace{\lfloor \cdots \lfloor \lfloor n/2 \rfloor /2 \rfloor \cdots /2 \rfloor}_{h \text{ floors}}$$

   We first prove that for any given n the following identity holds:

   $$\left\lfloor \frac{\left\lfloor \frac{n}{2^{h-1}} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2^h} \right\rfloor$$

   Proof. Consider any integer n in its binary form. When we divide n by 2^{h-1}, we are effectively right-shifting the binary representation of n by $h-1$ bits. The floor operation simply removes any fractional part that may arise from this division. Now, when we take the result and divide it by 2 again (which is equivalent to right-shifting by one more bit), we are effectively right-shifting the original binary representation of n by a total of h bits. The floor operation again removes any fractional part. Therefore, the two sides of the equation represent the same operation on the binary representation of n, leading to the same result.

   Hence, we have:

   $$\left\lfloor \frac{\left\lfloor \frac{n}{2^{h-1}} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2^h} \right\rfloor$$

   □

   Using this identity, we can simplify $n^{(h)}$ as follows:

   $$n^{(h)} = \left\lfloor \frac{n}{2^h} \right\rfloor$$

Therefore, the number of nodes at height h is $\left\lceil \frac{\lfloor \frac{n}{2^h} \rfloor}{2} \right\rceil$.

We then prove that $\left\lceil \frac{\lfloor \frac{n}{2^h} \rfloor}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$:

Proof. Since $\lfloor x \rfloor \leq x$ for any real number x, we have

$$\frac{\lfloor \frac{n}{2^h} \rfloor}{2} \leq \frac{\frac{n}{2^h}}{2} = \frac{n}{2^{h+1}}$$

Taking the ceiling of both sides, we get

$$\left\lceil \frac{\lfloor \frac{n}{2^h} \rfloor}{2} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

□

3.

```
function ITERATIVE-MAX-HEAPIFY(A, i)
    // A is a max-heap, 1-indexed.
    // i is the index to heapify.
    // No output since A is modified in place.
    j ← i
    while true do
        L ← 2*j
        R ← 2*j + 1

        largest ← j
        if L ≤ A.heap_size and A[L] > A[largest] then
            largest ← L
        fi
        if R ≤ A.heap_size and A[R] > A[largest] then
            largest ← R
        fi

        if largest = j then
            return
        fi

        swap A[j], A[largest]
        j ← largest
    end
end
```

Listing 1: Iterative Max-Heapify

4.

```
function DELETE(A, i)
    // A is a max-heap, 1-indexed.
    // i is the index to delete.
    // No output since A is modified in place.
    require 1 ≤ i ≤ A.heap_size

    if i = A.heap_size then
        A.heap_size ← A.heap_size - 1
        return
    fi
```

```

    swap A[i], A[A.heap_size]
    A.heap_size ← A.heap_size - 1
    if i > 1 and A[i] > A[PARENT(i)] then
        // Bubble up
        HEAP-INCREASE-KEY(A, i, A[i])
    else
        // Sift down
        MAX-HEAPIFY(A, i)
    fi
end
```

Listing 2: Delete

- **Why it works:** Swapping the target with the last element removes the target after shrinking the heap. Only position i can violate the heap property. If the new key at i exceeds its parent, raising it with HEAP-INCREASE-KEY restores order on the path to the root, otherwise MAX-HEAPIFY restores order on the path to leaves.
- **Runtime:** Both HEAP-INCREASE-KEY and MAX-HEAPIFY take $O(\log n)$ time. Since either one of them is called in DELETE, the runtime of DELETE is $O(\log n)$.

5.
```
function MULTIMERGE(A)
    // Input: A, an array of k sorted arrays in ascending order, each of length n/k.
    // Output: B, the sorted merge of all arrays in A.
    k ← A.length
    n ← k * A[1].length

    H ← empty array of size k as min-heap
    for i ← 1 to k do
        // Insert the first element of each array into the heap.
        MIN-HEAP-INSERT(H, A[i][1], k)
    end

    B ← new array of length n
    p ← 1
    while H.heap_size > 0 do
        u ← MIN-HEAP-EXTRACT-MIN(H)
        B[p] ← u
        p ← p + 1

        // Insert the next element if necessary.
        if u.id_in_arr < A[u.arr_id].length then
            next_element ← A[u.array_index][u.index_in_array + 1]
            MIN-HEAP-INSERT(H, next_element, k)
        end
    end

    return B
```

Listing 3: Multimerge

6. *Proof.* Notice that

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(1) + 2 + 3 + \cdots + (n-1) + n \\ &= T(0) + 1 + 2 + 3 + \cdots + (n-1) + n \\ &= T(0) + \frac{n(n+1)}{2} \in \Theta(n^2) \end{aligned}$$

□

7. We substitute $T(n) = T(2^k)$ with $S(k)$. Then the recurrence becomes

$$S(k) = 2S(k-1) + 2^k \cdot k, \quad S(0) = 1.$$

Divide both sides by 2^k:

$$\frac{S(k)}{2^k} = \frac{S(k-1)}{2^{k-1}} + k.$$

Let $P(k) = \frac{S(k)}{2^k}$. Then we have

$$P(k) = P(k-1) + k, \quad P(0) = 1.$$

From the result of Problem 6, we know that

$$\begin{aligned} P(k) &= P(0) + \frac{k(k+1)}{2} \\ &= 1 + \frac{k(k+1)}{2} \end{aligned}$$

Hence,

$$S(k) = 2^k \cdot P(k) = 2^{k-1}\left(k^2 + k + 2\right).$$

Substitute back to n:

$$T(n) = \frac{n}{2}\left(\log^2 n + \log n + 2\right).$$