Exercise on Heap

$44251017 \ HuangJiahui$

(1) Show the bottom-up construction of a heap over keys 5, 3, 8, 7, 2, 6. Depict intermediate heaps.

> **Input keys:** 5, 3, 8, 7, 2, 6
>
> **Initial array representation (1-indexed):**
> `[ , 5, 3, 8, 7, 2, 6]`
>
> **Bottom-up heap construction steps:**
>
> Start from index $\lfloor n/2 \rfloor$ = 3 and perform downheap from index 3 to 1.
>
> - **At index 3 (value = 8)**
>   → child at index 6 = 6 → 6 < 8 → **swap**
>   → `[ , 5, 3, 6, 7, 2, 8]`
> - **At index 2 (value = 3)**
>   → children: 7 (index 4), 2 (index 5)
>   → 2 is smallest → **swap 3 and 2**
>   → `[ , 5, 2, 6, 7, 3, 8]`
> - **At index 1 (value = 5)**
>   → children: 2 and 6 → 2 is smallest → **swap**
>   → `[ , 2, 5, 6, 7, 3, 8]`
>   → 5 at index 2, children = 7, 3 → swap 5 with 3
>   → `[ , 2, 3, 6, 7, 5, 8]`
>
> Final heap (array): `[ , 2, 3, 6, 7, 5, 8]`

---

(2) Suppose that heap H1 has n keys and heap H2 has m keys. Now show an efficient algorithm that generates heap H3 consisting of the keys contained in both H1 and H2. Namely, H3 consists of the intersection of the keys of H1 and H2.

```
  S ← empty hash set                          // To store keys from H1 for fast lookup

  for each key x in H1:
    S.add(x)                                  // Store all keys of H1 in the set

```

```
  L ← empty list                              // List to collect common keys

  for each key y in H2:
    if S.contains(y):                         // Check if y exists in H1
      L.add(y)                                // If yes, add to list

  H3 ← BottomUpHeapConstruct(L)               // Build heap from intersection list

  return H3
```

(3) Show the worst case running of your algorithm.

Let us analyze the worst-case running time of the above algorithm step by step.

Step-by-step Cost Breakdown:

```
1. Insert all elements from H1 into a set:         O(n)
2. Iterate through H2 to find common keys:         O(m)
3. Build heap H3 from k intersected keys:          O(k)
```

Where:

- n = size of H1
- m = size of H2
- k = number of keys in the intersection (k ≤ min(n, m))

If all elements in H1 and H2 are distinct, `k = 0` → heapify takes $O(0)$
If all elements match, `k = min(n, m)` → heapify takes $O(min(n, m))$

Thus, the worst case time = $O(n + m + k) = O(n + m)$