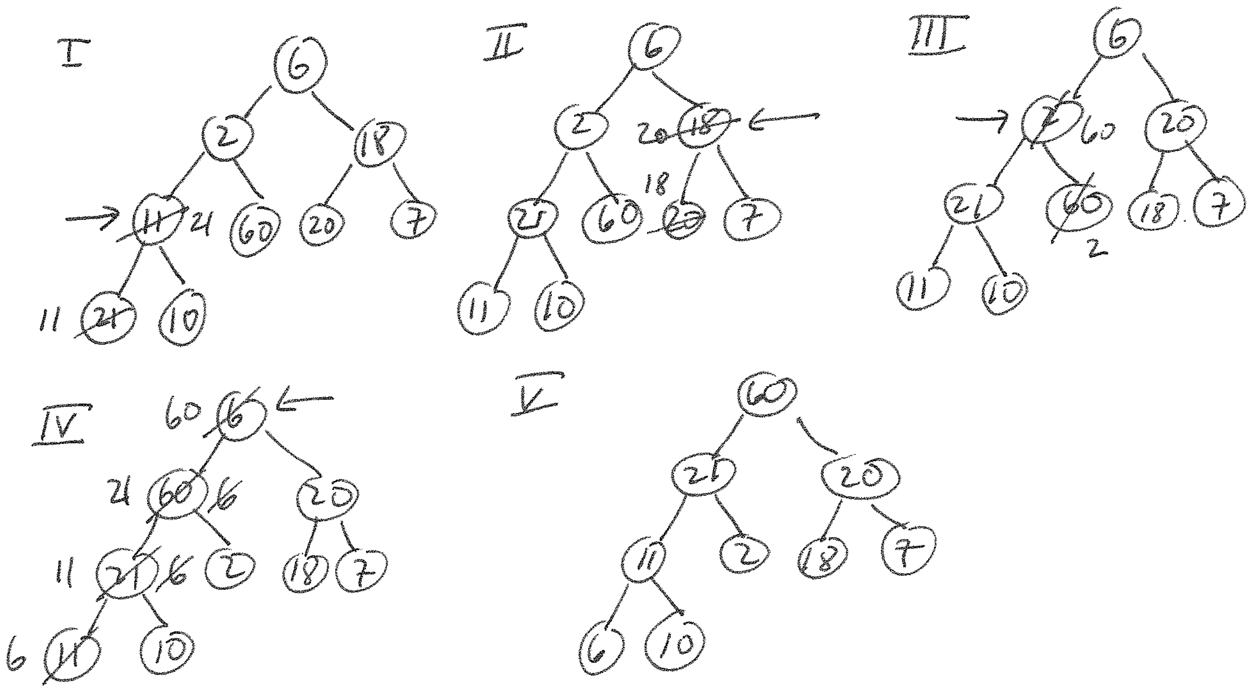
CS 584 HW #1 Key

1A $3^{n+2} = 3^2 \cdot 3^n = 9 \cdot 3^n \le \underbrace{9}_{c} \cdot 3^n$ for $n > 0$

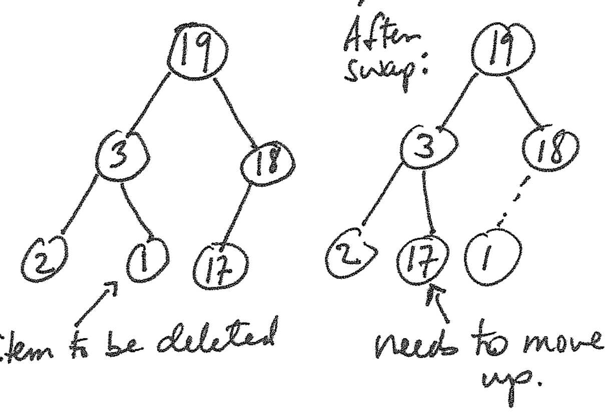
Hence $3^{n+2} \in O(3^n)$

$3^{2n} = 3^n \cdot 3^n$. Suppose $3^n \cdot 3^n \le c \cdot 3^n$ for some c and $n > n_0$. That would mean $3^n \le c$ for $n > n_0$. However, note that $3^k > k$ for $k \ge 0$. So choose $j \ge \max(c, n_0)$, and $3^j > j \ge c$. So no such c exists and $3^{2n} \notin O(3^n)$.

1B.



1C. The strategy is to swap the deleted element with the last element in the heap, then remake the heap. A critical observation is that the moved element may need to move upward in the heap to get to the correct position



B

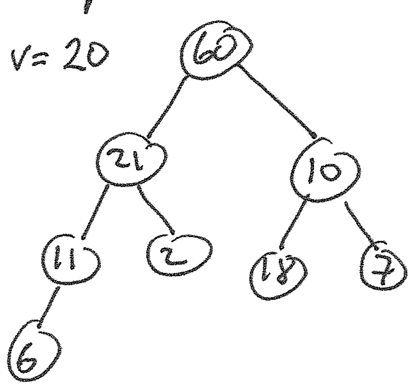
There is an existing procedure, Heap-Increase-Key that will "bubble" an element upward. If the new element at A[i] is greater than the previous element, we ~~will~~ call that procedure. If not, the new element may need to move downward to restore the heap, which Max-Heapify will do.

```
Heap-Delete (A, i)
    v ← A[i]
    A[i] ← A[heapsize[A]]
    heapsize[A] ← heapsize[A] - 1
    if A[i] > v
        then Heap-Increase-Key (A, i, A[i])
        else Max-Heapify (A, i)
```

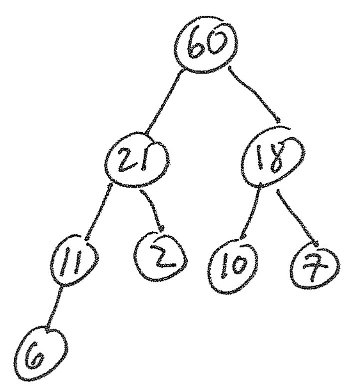
Note that whichever branch of the if is taken, the amount of work is bounded by the height of the heap, which is $O(\lg n)$.

Example: Suppose we call Heap-Delete (A, 3) where A is as shown in V. of the answer to 1B. Then after the swap and shortening of the heap, we have:

$v = 20$



since $10 \leq 20$, we call Max-Heapify (A, 3) to get:



10. (a) $f(n)$ in $O(g(n))$ does not imply $g(n) \in O(f(n))$ $\boxed{C}$

Counterexample: $f(n) = n$, $g(n) = n^2$.

(b) $f(n)$ in $O(g(n))$ implies $O(f(n) + g(n)) = O(g(n))$

- Clearly $O(g(n)) \subseteq O(f(n) + g(n))$, since any function $h(n) \leq c \cdot g(n)$ for $n > n_0$, we also have $h(n) \leq c.(f(n) + g(n))$ for $n > n_0$.
- Suppose $h(n) \in O(f(n) + g(n))$. Then there is a $d > 0$ and an n_1 such that $h(n) \leq d.(f(n)+g(n))$, $n > n_1$. Since $f(n) \in O(g(n))$, there is a $e > 0$ and an n_2 such that $f(n) \leq e.g(n)$ for $n > n_2$.
- Let $b = d.(e+1)$ and $n_3 = \max(n_1, n_2)$. Then
  $b \cdot g(n) = d.(e.g(n) + g(n)) \geq d.(f(n) + g(n)) \geq h(n)$.
  So $h(n) \in O(g(n))$. Since $h(n)$ was arbitrary, $O(f(n) + g(n)) \subseteq O(g(n))$. Hence $O(f(n)+g(n)) = O(g(n))$.

(c) $f(n)$ in $O(g(n))$ does not imply $2^{f(n)}$ in $O(2^{g(n)})$.

Counter example $f(n) = 2n$, $g(n) = n$. Similary to 1-A, we know that $2^{2n} \notin O(2^n)$.

(d) $f(n) \in O(g(n))$ implies $\lg f(n) \in O(\lg g(n))$.

If $f(n) \in O(g(n))$, then there are $c > 0$ and n_0 with $f(n) \leq c.g(n)$ for $n > n_0$. Since $\lg$ is an increasing function, the inequality is preserved if we take logs of both sides: $\lg f(n) \leq \lg(c.g(n)) = \lg c + \lg g(n)$ for $n > n_0$

Let $d = \lg c + 1$. Then $d.(\lg g(n)) = (\lg c).(\lg g(n)) + \lg g(n)$
$\geq \lg c + \lg g(n) \geq \lg f(n)$ for $n > n_0$. So
$\lg f(n) \in O(\lg g(n))$. [I guess this assumes $g(n) \geq 2$ for $n > n_0$].

1E. The list elements need to be turned into (key, list#) pairs. [D]

[There can also be a pointer to ~~a~~ the record, if it is longer than just the key.] Note that each component comes from a bounded range: $1 \le key \le m$, $1 \le list\# \le k$, which should suggest something like counting sort or bucket sort, which are $O(l+r)$ for l elements in the range $1..r$.

Pseudocode

1. Create a list L of pairs (i,j) where i is ~~the~~ the key of a record in list j. $O(k \cdot n)$
2. Bucket sort L on the key component $O(k \cdot n + m)$
3. * Bucket sort L on the list# component $O(k \cdot n + k) = O(k \cdot n)$
4. Each bucket contains a sorted list in its first components $O(k \cdot n)$

* The sort used needs to ~~be~~ be stable: it ~~keeps~~ keeps elements with the same list# in the same order on key.

Complexity: $O(k \cdot n + m)$ is the dominant cost.

Example: $m = 6$, $k = 3$, $n = 4$

$l_1 = 1, 6, 4, 4$ $l_2 = 2, 5, 2, 4$ $l_3 = 3, 2, 4, 6$

$L = (1,1), (6,1), (4,1), (4,1), (2,2), (5,2), (2,2), (4,2), (3,3), (2,3), (4,3), (6,3).$

Bucket sort 2.

1 → (1,1)
2 → (2,2) → (2,2) → (2,3)
3 → (3,3)
4 → (4,1) → (4,1) → (4,2) → (4,3)
5 → (5,2)
6 → (6,1) → (6,3)

Bucket sort 3.

1 → (1,1) → (4,1) → (4,1) → (6,1)
2 → (2,2) → (2,2) → (4,2) → (5,2)
3 → (2,3) → (3,3) → (4,3) → (6,3)

Sorted lists

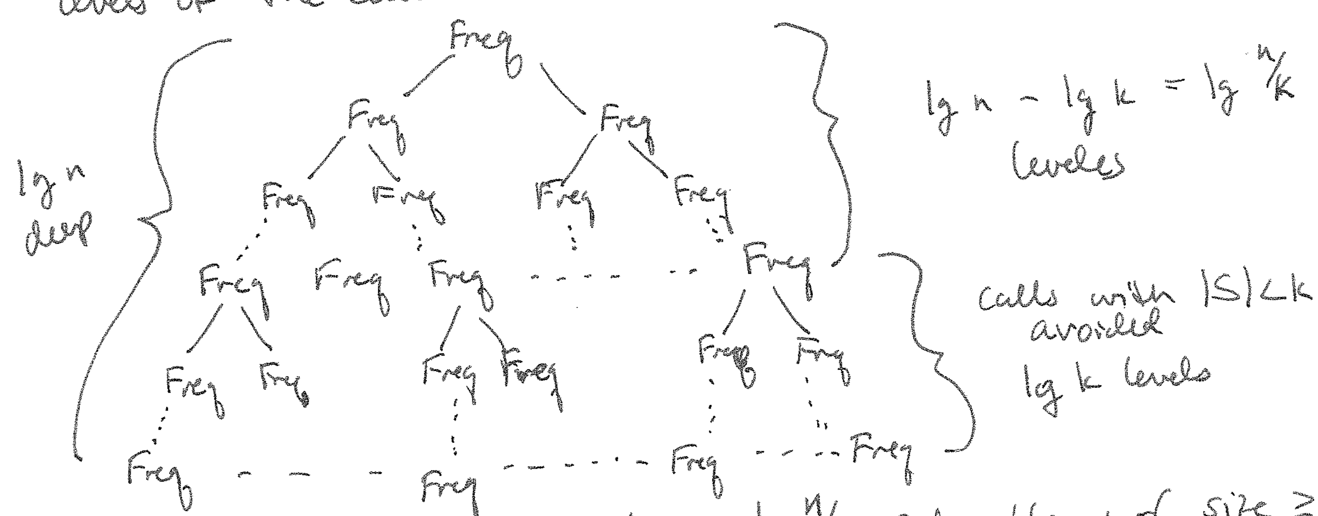
$l_1 = 1, 4, 4, 6$

$l_2 = 2, 2, 4, 5$

$l_3 = 2, 3, 4, 6$

1-F There are two changes to the Freq algorithm. One is that we always ~~s~~ use Median(S) to choose e for splitting S into S_low and S_high. With just this change, we are guaranteed an $O(n \lg n)$ algorithm, since the depth of recursion is at most $\lg n$, and the combined sizes of sub-problems at each level is $< n$. |E|

The second change is to always work on the largest remaining subproblem first, and never call a subproblem of size less than the ~~best~~ ~~median~~ most frequent item found so far. This policy guarantees that we will discover m before we call any problems of size $\leq k$. We can use a priority queue (say, implemented as a heap) to find the largest remaining subproblem at any point. The policy cuts off any recursive calls of size $< k$, hence the bottom ~~k~~ $\lg k$ levels of the call tree are avoided.



Also note that there are at most n/k subproblems of size $\geq k$, so the heap never gets deeper than $\lg n/k$ levels. So any Extract-Min or Insert is $O(\lg n/k)$.

Since the combined size of problems at each level is still $< n$, we get the $O(n \lg n/k)$ bound on time.

$\boxed{F}$

Example (i) = order of subproblems