1. At what positions of a heap might the largest key be stored? Provide a statement and proof. Hint: use contradiction.

**Solution.**

**Theorem 1.** The largest key of a minimum heap will be stored at the “bottom” level of the heap. Bottom means that both of its children are leaf nodes of the heap.

**Proof.** Assume it was stored at a different level of the tree. Then, one of its children would store a key (be an internal node also). But, based on the heap property that each child node must have a key greater than or equal to its parent. This contradicts the fact that we have a heap and cannot be possible, thus the largest key of a minimum heap will be at the bottom.

2. Bill claims that a preorder traversal of a heap will list its keys in nondecreasing order. Provide a counter proof (a counter proof means a contradicting example) of this claim. Use an image to aid your proof.

**Solution.**

Figure 1 shows a counter example to Bill’s claim.

![Figure 1](image)

Figure 1: Heap that shows counter example for a standard preorder traversal.
3. Tamarindo Airlines wants to give a first-class upgrade coupon to their top \( \log n \) frequent fliers, based on the number of miles accumulated, where \( n \) is the total number of the airline’s frequent flyers. The algorithm they currently use, which runs in \( O(n \log n) \) time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top \( \log n \) flyers. Describe and analyze an algorithm that identifies the top \( \log n \) flyers in \( O(n) \) time. Hint: solve the next problem first.

### Algorithm.

The key idea is to use a maximum heap. Then, use bottom-up heap construction to compute a heap in \( O(n) \) time. After, remove \( \log n \) frequent fliers from the list. This process is shown in Algorithm 1.

#### Algorithm 1 Top \( \log n \) Frequent Fliers

**Input:** \( n \) frequent fliers as an array \( A \)

**Output:** List \( L \) of \( \log n \) top frequent fliers

1. List \( L \leftarrow \emptyset \)
2. Maximum Heap \( H \leftarrow \text{heapify}(A) \)
3. for \( i \leftarrow 1 \ldots \log n \) do
4. \( \text{L.addLast}(H.removeMin()) \)
5. return \( L \)

### Time Complexity.

**Theorem 2.** Algorithm 1 runs in \( O(n) \) time.

**Proof.** Heapify takes \( O(n) \) time. Then, clearly there are \( \log n \) removals from the maximum heap. A removal from a heap takes \( O(\log n) \) time. \text{addLast()} of a list can be implemented in \( O(1) \) time with a good implementation. Thus, in total the complexity is \( O(n + \log n \log n) = O(n) \).

### Memory Complexity.

**Theorem 3.** Assuming \text{heapify()} happens in-place of the collection, Algorithm 1 can be implemented using \( O(1) \) extra memory.

**Proof.** Using an in-place maximum heap of the collection means using no additional memory. Items are simply swapped in the original memory. So, because only a constant number of temporary variables is needed for these swapping operations, \( O(1) \) excess memory is required.

**Note:** \( O(n) \) is also correct if you do not assume an in-place heapify.
4. Explain how the $k$ largest elements from an unordered collection of size $n$ can be found in time $O(n + k \log n)$ using a maximum-oriented heap.

**Algorithm.**

The key idea to the algorithm is as follows. Create a maximum heap from the elements of the collection. If needed, the collection may be preprocessed to be stored as an array. Then remove $k$ items. The algorithm is shown in Algorithm 2.

**Algorithm 2 $k$-largest**

| Input: | Collection $C$, integer $k$ |
| Output: | List $L$ of $k$-largest elements |

1: List $L \leftarrow \emptyset$
2: Maximum Heap $H \leftarrow \text{heapify}(C)$
3: for $i \leftarrow 1 \ldots k$ do
4: $L.$addLast($H.$removeMin())
5: return $L$

**Time Complexity.**

**Theorem 4.** Algorithm 2 runs in $O(n + k \log n)$ time.

*Proof.* Heapify takes $O(n)$ time. Then, clearly there are $k$ removals from the maximum heap. A removal from a heap takes $O(\log n)$ time. $\text{addLast()}$ of a list can be implemented in $O(1)$ time with a good implementation. Thus, in total the complexity is $O(n + k \log n)$.

**Memory Complexity.**

**Theorem 5.** Assuming $\text{heapify()}$ happens in-place of the collection, Algorithm 2 can be implemented using $O(1)$ extra memory.

*Proof.* Using an in-place maximum heap of the collection means using no additional memory. Items are simply swapped in the original memory. So, because only a constant number of temporary variables is needed for these swapping operations, $O(1)$ excess memory is required.

Note: $O(n)$ is also correct if you do not assume an in-place heapify.
5. **Bonus.** A group of children want to play a game, called Unmonopoly, where in each turn two things happen. (1) A player takes a turn that changes their amount of money, (2) the player with the most money must give half of their money to the player with the least amount of money. What data structure(s) should be used to play this game efficiently? Justify your choice through some basic analysis.

The key idea here is two maintain three data structures. First, have an array list of player objects. Second, have a minimum heap of player objects. Third, have a maximum heap of player objects. Now, each of the heaps are locator based adaptable priority queues who’s key is the amount of money the player has. Each player object stores an index of where it sits in the array list, a locator to its position in the minimum heap, and a locator to its position in the maximum heap.

During each turn, the $i^{th}$ player updates his amount. The player at the root of the maximum heap gives half of his money to the root of the minimum heap. Now three players have had updates to their money and thus the properties of the heaps are broken. So there will be either an upheap or a downheap required to maintain each of the players in each of the heaps (six updates in total).

The time complexity of this operation takes $O(\log n)$. There will be altering money amount that happens in constant time. Accessing each players location in the heaps happen in $O(1)$ time because of the array list and locator-based implementation. Then the 6 upheaps or downheaps all happen in $O(\log n)$.

This is optimal because there is no way to maintain both the maximum and minimum of a set of data and be able to guarantee $O(\log n)$ performance of updating the amounts of money.