Problem 6.4

Insert Algorithm

```
function INSERTBUILD(pairs)
   input:  array of KVpairs
   output:  min heap containing each KVpair
   heap ← new array[size of pairs]
   for all pair ∈ pairs do
      insert pair in heap
   end for
end function
```

Sift Down Algorithm

```
function SIFTDOWNBUILD(pairs)
   input:  array of KVpairs
   output:  min heap containing each KVpair
   heap ← new array[size of pairs]
   for int i from 0 to (size of heap)/2 - 1 do
      SIFTDOWN(heap[i])
   end for
end function
```

Analysis for insertBuild

INSERTBUILD runs in loglinear time. For all n pairs in the list inputted, it must perform an insert operation, which means it must call SIFTUP on every pair as well. In the worst case, SIFTUP's runtime is logarithmic with respect to the size of the heap at the time it is called. For the first few inserts, the size is much smaller than n. However, in a balanced tree, the entire second half of the pairs will be inserted when the tree is at its full depth. Therefore, for $n/2$ pairs, this algorithm calls a function with complexity $O(\log n)$, making the total complexity of the algorithm $O(n \log n)$

Analysis for siftDownBuild

siftDownBuild runs in linear time. The intuition for why siftDown is faster that siftUp is that, while both do run in logarithmic time given an arbitrary slot in the tree, siftDown only performs at worst the number of swap operations equal to the depth below the level of the pair being siftDown-ed. On the other hand, siftUp performs the number of swaps equal to the depth above the relevant pair. This is significan because, in a balanced tree, a full half (plus one) of the pairs are located in the bottom row. While this means, for insertBuild, that half of the calls to siftDown will be maximally costly, for siftDown-Build half the operations will be maximally cheap. In fact, in the algorithm I suggest for siftDownBuild, the entire bottom row of the tree need not be traversed because there is nowhere below these nodes to siftDown to. Thus the algorithm starts at the first node and iterates until the last node with daughters, which is index $\frac{n}{2} - 1$.

Below is a slightly more formal demonstration of the runtime of siftDownBuild. I assume a balanced heap and define the cost of a constant time swap as 1.

level	# of nodes	# of swaps/node	# level cost
0	1	$\lfloor \log_2 n \rfloor$	$\lfloor \log_2 n \rfloor$
1	2	$\lfloor \log_2 n \rfloor - 1$	$2(\lfloor \log_2 n \rfloor - 1)$
2	4	$\lfloor \log_2 n \rfloor - 2$	1 $4(\lfloor \log_2 n \rfloor - 2)$
3	8	$\lfloor \log_2 n \rfloor - 3$	$8(\lfloor \log_2 n \rfloor - 3)$
...	...	...	...
i	2^i	$\lfloor \log_2 n \rfloor - i$	$2^i(\lfloor \log_2 n \rfloor - i)$
...	...	...	...
$\lfloor (\log_2 n) \rfloor - 1$	$\frac{n}{2}$	1	$\frac{n}{2}$

The total cost of siftDownBuild is the sume of all the level costs. This sum can be expressed as:

$$\sum_{i=0}^{(\log_2 n)-1} 2^i(\lfloor \log_2 n \rfloor - i)$$
$$= 2n - \log_2 n - 2$$

sum calculated by WolframAlpha

Since log terms are lower order than linear terms, the total cost is an element of $O(n)$.