Searching / sorting

Q1.

1) quicksort:

1$\underline{2}$55736
123$\underline{5}$576
1235$\underline{5}$76
123556$\underline{7}$

2) insertion sort:

$\underline{2}$557316
2$\underline{5}$57316
25$\underline{5}$7316
255$\underline{7}$316
2$\underline{3}$55716
$\underline{1}$235576
12355$\underline{6}$7

3) mergesort:

```
255  7316
2 55 7316
2 5 5 7316
2 55 7316
255 7316
255 73 16
255 7 3 16
255 37 16
255 37 1 6
255 37 16
255 1367
1235567
```

Q4.

(a) All equal. Insertion: O(n), Merge: O(n log n), Quicksort: O(n^2)
(b) In order. Insertion: O(n), Merge: O(n log n), Quicksort: O(n^2), ~O(n log n) if optimized with medianOf3 for pivot.
(c) Reverse Order. Insertion: O(n^2), Merge: O(n log n), Quicksort: O(n^2), ~O(n log n) if optimized with medianOf3 for pivot.

Algorithmic Analysis

Q1.

(a)

O(2n log n + 4n + 17 log n) = O ((2n + 17) log n + 4 n) = O (n log n + n)
O(log (3n)) = O (log n + log 3) = O (log n)

(b) No, since O (log (n-1)) is equal to O (log (n)), there is no significant performance gain for sufficiently large n's.

(c)

There was some dispute over what "m" signifies for this problem. If m is the _value_ of the item we are looking for, then the complexity of the algorithm is O(n) since we must potentially visit every element in the list. If m is the _index_ of the item we are looking

for, then the runtime is O(m) since we know we must visit m-1 elements before we find our value.
Q2.
sorted array. Insert: O(log n + n) (binary search and copy). Find: O (log n) (binary search) getMin: O(1) (first element is smallest) successor: O(log n) (binary search, add one)

unsorted array. Insert: O(n) (scan from 0..m, copy m+1 to n) find: O(n) (linear scan) getMin: O(n) (linear scan) or could be O(1) if we keep track of the minimum on each insert/delete successor: O(n) (linear scan)

hashtable. Insert: O(1) (if load factor is maintained) find: O(1) (if load factor is maintained) getMin: O(n) (scan through all elements) successor: O(n) (scan through)

sorted linked list: insert: O(n) (scan, no copy necessary, though, so actual running time will be much better than array) find: O(n) (scan through list, cannot do binary search on a linked list)
getMin: O(1) (first element in list) successor: O(n) (scan through, get next element)

unsorted linked list: insert: O(n) (scan, no copy necessary, though, so actual running time will be much better than array) find: O(n) (scan through list)
getMin: O(n) (scan through list) successor: O(n) (scan through, get next element)

Data structures:

Q2.

0	1	2	3	4	5	6	7	8	9	10
ibex	bat	hare	koala	Dog		ape	mud			

Q4.
Right most element(40)

(a) pre 22 14 11 16 25 23 24 40 32
in 11 14 16 22 23 24 25 32 40
post 11 16 14 24 23 32 40 25 22

(b) after inserting 15
15 becomes the left child of 16

Q5.
```
public mp(tnode r, int k1, int k2, boolean printValue){
        if(r.val==k1){printValue=true;}
        if(r.val==k2){return;}
        mp(r.left, k1,k2,printValue);
        if(printValue){System.out.print(r.val);}
        mp(r.right, k1,k2, printValue);
}
```
We traverse the tree until we see k1, at which point we start printing values out until we see k2.

Q7.

(a) For array: Insert takes O(n) (scan to 1..m, insert and copy m+1..n), DeleteMax takes O(1) (if we save the index, remove that element from the end), DeleteMin O(n) (Remove first element, copy n-1 elements over to the left)
For DLL: Insert takes O(n), DeleteMax takes O(1) (if it's a circular list, we can just scan back from the head to get the max), DeleteMin O(1)
Doubly linked list is better, especially because while the worse case running time for insert are the same for list and array, O(n), in practice the insert into a linked list is quicker since there is no need to "copy" the tail of the linked list, we just set the appropriate references. Ie, insert on a sorted array will always take exactly O(n) time, but a linked list will take O(m) time where m is the index of the element being inserted. (m <= n)

(b)
You can look up a keyword with both data structures in O(log n) time, however, inserting a new keyword into the set is much faster (O(log n)) with the tree than with the array (O(n)). So, the tree works best here.

(c)
For a good hash function and well implemented hash table, insert and find can be made to be constant time. So, you can't do better than that, and the hashtable wins.

(d) For the hashtable, insert is constant time. For the array (if we insert at an arbitrary place) it's O(n). FindMax and FindMin both take O(n) time for hashtable and unordered array since we have to do a scan through the whole thing. So, the hashtable wins due to its good insertion speed.