3.1. Quicksort fused with insertion sort

1) If we use quicksort on a sub-arrays with $\leq k$ size, when recursion will approach $\frac{n}{k}$ level, the expected running time of this process will take $O(n \log \frac{n}{k})$ (since expected running time of quick-sort is $O(n \log n)$).
After the top-level call to quicksort returns, we use insertion sort on the entire array. But because we already used quicksort on $\leq k$ elements, there are at most k unsorted elements. Therefore, insertion sort will check at most k elements, and the expected running time of this part will be $O(nk)$
By combining two parts, we get expected running time $O(nk + n \log \frac{n}{k})$

2) Theoretically, we should pick such k, that

$$nk + n \log \frac{n}{k} \leq \underbrace{n \log n}_{\text{(avg of quicksort)}} \; | : n$$

$$k + \log \frac{n}{k} \leq \log n$$

$$k + \log n - \log k \leq \log n$$

$$k \leq \log k$$

There are no real solutions, and only in practice it will be possible to find out good values for k.

3.2. BST with equal keys

1. If we will use regular insertion algorithm, then, depend in on the implement-

-ation, the tree will grow only either to the left or to the right, forming the height $\Theta(n)$ and time complexit $\Theta(n^2)$

2. With this strategy, the tree will be balanced, and after 2^k insertions the tree will be complete, with the height $\Theta(\log n)$ and overall time complexity $\underline{\Theta(n \log n)}$

3. Since the height of the tree will always be $\Theta(1)$, the complexity is $\Theta(n)$

4. The chance is 50/50, therefore, if we are unlucky enough in worst-case the tree will be unbalanced with height $O(n)$ and complexity $\underline{O(n^2)}$

However, in average the tree will be almost balanced with height $O(\log n)$ and complexity $O(n \log n)$

3.3. d-ary heaps

1. Similarly to binary heap.

The parent of i-th element will be:

$(i-2)/d + 1$

the child of i-th element will be:

$j + 1 + (i-1) \cdot d$

2. Each node has d children, therefore, the height will be $\log_d n$

3. The extractMax stays the same in d-ary heap, as in binary heap.
Implementation taken from Cormens' book for binary heap:

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Worst-case: $O(d \log_d n)$

4. The insert will also stay the same:

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

To insert, we increase current heap size by 1 and then call increase-key.
Worst-case: $O(\log_d n)$