1. The idea for this algorithm is quite simple. If you are looking for the $i$th smallest key, there are three cases depending on the size, $k$, of the left subtree of the root (i.e., the number of keys smaller than the root’s key): if $k = i - 1$ then the root is the $i$th smallest; if $k > i - 1$ then the $i$th smallest is in the left subtree of the root and it is the $i$th smallest in that subtree; if $k < i - 1$ then the $i$th smallest is in the right subtree of the root and it is the $i - 1 - k$th smallest in that subtree. The Java code might look something like:

```java
public static Object select(BinaryTreeNode node, int i) {
    if (node == null) return null;
    if (node.left() == null) {
        int leftsize = 0;
    } else {
        int leftsize = ((BinaryTreeNode)node.left()).size();
    }
    if (leftsize == i - 1) return node.value();
    if (leftsize > i - 1) return select(node.left(), i);
    if (leftsize < i - 1) return select(node.right(), i - 1 - leftsize);
}
```

2.
3. We use the cost formulas for unordered chaining: search hit has cost $(1 + \alpha)/2$; search miss has cost $\alpha$; where $\alpha$ is $n/m$. (Note the formulas in the book are different (wrong!) but you can use them as well.) We get the following for the expected cost of hits and misses in the two cases:

(a) In this case $\alpha = 10$ so that a search hit has cost 5.5 and a search miss has cost 10.

(b) In this case $\alpha_1 = 4$ (for the first table) and $\alpha_2 = 16$ for the second table. For a search hit the expected cost is $4/5 * (1 + \alpha_1)/2 + 1/5 * (\alpha_1 + (1 + \alpha_2)/2) = 4.5$. Note the second $\alpha_1$ for the case of a search miss in the first table. For a search miss the expected cost is $\alpha_1 + \alpha_2 = 20$.

Overall there is a small gain for hits but a big loss for misses.

4. Consider the key 0 and the key $2^i$ ($0 \leq i \leq k - 1$). These two keys differ in the $i$ bit position only. (Here bit positions are labelled 0 through $k - 1$ starting with the least significant bit.) The key 0 is equal to 0 mod $M$ so it gets mapped to position 0 in the array but $2^i \mod M$ is always non-zero (since if $M$ is a prime greater than 2 it will never evenly divide $2^i$) and therefore is not mapped to position 0, i.e., they are mapped to different places. (Note: the statement doesn’t hold for $M = 2$.)

5. For key $(a, b)$ use the hash function $h(a, b) = 13 * a + b.$