1. When separate chaining is used, the hash table will look like this:

2. Linear probing:

3. Double hashing:

4. We solve the equation using repeated substitution:

\[
\begin{align*}
f(n) &= f(n-1) + c_2n + c_3 \\
f(n-1) &= f(n-2) + c_2(n-1) + c_3 \\
f(n-2) &= f(n-3) + c_2(n-2) + c_3 \\
&\vdots \\
f(3) &= f(2) + c_2 \times 3 + c_3 \\
f(2) &= f(1) + c_2 \times 2 + c_3 \\
f(1) &= f(0) + c_2 + c_3 \\
f(0) &= c_1
\end{align*}
\]

Substituting the value of \(f(1)\) in the equation for \(f(2)\), then \(f(2)\) in the equation for \(f(3)\), and so on we get

\[
f(n) = \frac{c_2}{2}n^2 + \left( c_3 + \frac{c_2}{2} \right) n + c_1
\]
Discarding constant terms, and then getting the larger function between \( n^2 \) and \( n \) we get that \( f(n) \) is \( O(n^2) \).

5.(i) **Algorithm maxValue \((r)\)**

**In:** Root \( r \) of a tree storing an integer value in each node

**Out:** The largest value stored in the nodes of the tree

```plaintext
if r.isLeaf then return r.value
else {
    max ← r.value
    for each child \( c \) of \( r \) do {
        val ← maxValue\((c)\)
        if val > max then max ← val
    }
    return max
}
```

5.(ii) • To compute the time complexity of the algorithm, first we ignore the recursive calls. In each invocation, the algorithm performs a constant number \( c \) of operations if the current node \( r \) is a leaf. If \( r \) is an internal node, then the else statement is performed. In each iteration of the for loop a constant number \( c' \) of operations are performed (ignoring recursive calls) and the loop is repeated \( \text{degree}(r) \) times. Hence, the total number of operations performed by the “for” loop is \( c' \times \text{degree}(r) \). Outside the “for” loop an additional constant number \( c'' \) of operations is performed, so for an internal node \( r \) the algorithm performs \( c' \times \text{degree}(r) + c'' \) operations.

To take into consideration the recursive calls we need to understand what their purpose is. Observe that the algorithm implements a traversal of the tree, as when called on a node \( r \) the algorithm performs one recursive call per child of \( r \); therefore, the effect of the recursive calls is to make the algorithm visit each node of the tree once. Hence, the algorithm performs one call per node and so, the total number of operations performed by the algorithm is

\[
\sum_{\text{leaves}\ u} c + \sum_{\text{internal nodes}\ u} (c' \times \text{degree}(u) + c'') = c \times \#\text{leaves} + c'' \times \#\text{internal nodes} + c' \sum_{\text{internal nodes}\ u} \text{degree}(u)
\]

\[
= c \times \#\text{leaves} + c'' \times \#\text{internal nodes} + c'(n - 1)
\]

Discarding constants, we get that the order of the time complexity is \( O(\#\text{leaves} + \#\text{internal nodes} + n) \). Since \( \#\text{leaves} + \#\text{internal nodes} = n \), the time complexity of the algorithm is \( O(n) \).

Another algorithm for solving the problem:

**Algorithm maxValue \((r)\)**

**In:** Root \( r \) of a tree storing an integer value in each node

**Out:** The largest value stored in the nodes of the tree

```plaintext
max ← r.value
for each child \( c \) of \( r \) do {
    val ← maxValue\((c)\)
    if val > max then max ← val
}
return max
```

The analysis of the time complexity of this algorithm is very similar to that for the first algorithm. The time complexity of this algorithm is also \( O(n) \).