Solution 1:

(a) See the figure below (a).

(b) See the figure above (b).
(c) Preorder: \(\langle a,b,c,f,d,g,h,e,i,j \rangle\). Postorder: \(\langle f,c,g,h,d,i,j,e,b,a \rangle\).
(d) Pre: \(\langle a,b,d,g,e,h,c,f,i,j \rangle\). In: \(\langle g,d,b,e,h,a,c,i,f,j \rangle\). Post: \(\langle g,d,h,e,b,i,j,f,c,a \rangle\).
(e) See the figure above (c).

Solution 2: Our procedure works recursively. Let us assume inductively that, given any node \(p\), \(\text{getHeight}(\text{Node } p)\) returns the maximum heights of the subtree rooted at \(p\) and all its siblings to its right. To get \(p\)’s height, we do \(1 + \text{getHeight}(p.\text{firstChild})\). To combine this with the heights of its siblings to the right, we take the maximum of \(p\)’s height and that of its sibling to the right. Thus we have the following:

```c
int getHeight(Node p) {
    if (p == null) return -1
    else return max(1 + getHeight(p.firstChild), getHeight(p.nextSibling))
}
```

An example is shown in figure below.
Solution 3: We determine the inorder successor as follows. If the right child link is non-null, the inorder successor is the leftmost node in its right subtree. We first move to the right child, and then follow left-child links until we can go no further (see the figure above (b)). On the other hand, if the right child link is null, this node is the rightmost node in the left subtree of its inorder successor. We follow parent links as long as these links lead us along right-child links. If we reach the root, then there is no inorder successor. Otherwise the parent of the last node on this chain is the inorder successor (see the figure above (c)). The running time is clearly proportional to the height of the tree.

```java
Node inorderSuccessor(Node p) { // return p's inorder successor
    if (p.right == null) { // p has no right subtree
        Node q = p.parent
        while (q != null && p == q.right) { // follow up right-child chain
            p = q; q = q.parent
        }
    } else { // p has a right subtree
        Node q = p.right
        while (q.left != null) { // find its leftmost node
            q = q.left
        }
    }
    return q
}
```

Solution 4: Let’s first give the solution to the simpler case, where the sizes decrease by at least a third.

\[
\begin{align*}
n_1 & \leq \frac{n_0}{3} = \frac{n}{3}, \\
n_2 & \leq \frac{n_1}{3} = \frac{n}{9}, \\
n_3 & \leq \frac{n_2}{3} = \frac{n}{27}, \\ & \vdots , \\
n_k & \leq \frac{n}{3^k}.
\end{align*}
\]

We know that \(n_L = 1\), since it contains just the root of the tree, and therefore

\[
1 = n_L \leq \frac{n}{3^L} \iff 3^L \leq n \iff L \leq \log_3 n = \frac{\log n}{\log 3}.
\]

Now, let’s consider the case for general \(\alpha\), where \(0 < \alpha < 1\).

\[
\begin{align*}
n_1 & \leq \alpha n_0 = \alpha n, \\
n_2 & \leq \alpha n_1 = \alpha^2 n, \\
n_3 & \leq \alpha n_2 = \alpha^3 n, \\ & \vdots , \\
n_k & \leq \alpha^k n.
\end{align*}
\]

We know that \(n_L = 1\), since it contains just the root of the tree, and therefore

\[
1 = n_L \leq \alpha^L n \iff \frac{1}{\alpha^L} \leq n \iff L \leq \log_{1/\alpha} n = \frac{\log n}{\log(1/\alpha)} = c\log n,
\]

where \(c = 1/\log(1/\alpha)\).

Solution 5: In each case, we will apply a token-based approach to analyze each run. Each operation collects \(\alpha\) tokens, and we will show that the number of tokens suffices to pay for all the operations in the run. We will then derive a bound on \(\alpha\).
(a) (Expansion case) Let $m$ denote the size of the array at the start of the run, which implies that the initial value of $n$ is $\frac{m}{2}$ (since the array is always half full at the start of a run).

Each time we perform an operation we assess it $\alpha$ tokens. One of these tokens will pay for the operation itself, allowing us to bank $\alpha - 1$ tokens.

In order to overflow the array, we need to induce a net increase of $m - n = m - \frac{m}{2} = \frac{m}{2}$ entries. This requires at least $\frac{m}{2}$ operations (all pushes), implying that we have banked at least $(\alpha - 1)\frac{m}{2}$ tokens before we get to the end of the run. The reallocation cost is the time needed to copy $m$ elements into the new array. Therefore, we need to set $\alpha$ so the number of banked tokens is at least this large, that is, $(\alpha - 1)\frac{m}{2} \geq m$. It follows that setting $\alpha = 3$ satisfies this, and this is the amortized cost in the expansion case.

(b) (Contraction case) Let $m$ denote the size of the array at the start of the run, which (as before) implies that the initial value of $n$ is $\frac{m}{2}$. As in the expansion case, we assess $\alpha$ tokens per operation, using one to pay for the operation and banking the remaining $\alpha - 1$.

In order to underflow the array, we need to induce a net decrease of $n - \frac{m}{4} = \frac{m}{4} - \frac{m}{4} = \frac{m}{4}$ entries. This requires at least $\frac{m}{4}$ operations (all pops), implying that we have banked at least $(\alpha - 1)\frac{m}{4}$ tokens before we get to the end of the run. The reallocation cost is the time needed to copy $\frac{m}{4}$ elements into the new array. Therefore, we need to set $\alpha$ so the number of banked tokens is at least this large, that is, $(\alpha - 1)\frac{m}{4} \geq \frac{m}{4}$. It follows that setting $\alpha = 2$ satisfies this, and this is the amortized cost in the contraction case.

Solution to the Challenge Problem: Suppose that the number of nodes decreases even faster. In particular, suppose that $n_{i+1} \leq \sqrt{n}$. Prove that there is a constant $c$ such that $L \leq c \lg \lg n$ (that is, the log of the log of $n$). Let’s assume we end at level $L$, where $n_L = 2$. Starting at level 0, we have

\[
n_1 \leq n_0^{1/2} = n^{1/2}, \quad n_2 \leq n_1^{1/2} = n^{1/4}, \quad n_3 \leq n_2^{1/2} = n^{1/8}, \quad \ldots, \quad n_k \leq n^{1/2^k}.
\]

We know that $n_L = 2$, since it contains just the root of the tree, and therefore

\[
2 = n_L \leq n^{1/2^L} \iff \log 2 \leq \log \left( n^{1/2^L} \right) \iff 1 \leq \frac{1}{2^L} \log n \iff 2^L \leq \log n \iff L \leq \log \log n
\]

So, in fact $c = 1$. 

3