**Algorithm 7.2**

Decrease codeword lengths indicated by the first element in *type*$[j]$, recursively accessing other lists if that first element is a package.

```
take_package$(j)$
1: set $x \leftarrow$ the element at the head of queue type$[j]$
2: if $x =$ "package" then
3:     take_package$(j+1)$
4:     take_package$(j+1)$
5: else
6:     set $c_x \leftarrow c_x - 1$
7: remove and discard the first elements of queues value$[j]$ and type$[j]$
```

used, then the $L$ bit vectors storing *type* in total occupy $n - \log_2 n + 1$ words of memory. That is, under quite reasonable assumptions, the space requirement of reverse package merge is $O(n)$.

If space is at a premium, it is possible for the reverse package merge algorithm to be implemented in $O(L^2)$ space over and above the $n$ words required to store the input probabilities [Katajainen et al., 1995]. There are two key observations that allow the further improvement. The first is that while each package is a binary tree, it is only necessary to store the number of leaves on each level of the tree, rather than the entire tree. The second is that it is not necessary to store all of the trees at any one time: only a single tree in each list is required, and trees can be constructed lazily as and when they are needed, rather than all at once. In total this *lazy reverse package merge* algorithm stores $L$ vertical cross-sections of the lists, each with $O(L)$ items, so requires $O(L^2)$ words of memory.

If speed is crucial when generating optimal length-limited codes, the runlength techniques of Section 4.6 on page 70 can also be employed, to make a *lazy reverse runlength package merge* [Turpin and Moffat, 1996]. The resulting implementation is not pretty, and no pleasure at all to debug, but runs in $O((r + r\log(n/r))L)$ time.

Liddell and Moffat [2002] have devised a further implementation, which rather than forming packages from the symbol probabilities, uses Huffman's algorithm to create the packages that would be part of a minimum-redundancy code, and then rearranges these to form the length-limited code. This mechanism takes $O(n(L_H - L + 1))$ time, where $L_H$ is the length of a longest unrestricted minimum-redundancy codeword for the probability distribution being processed. This algorithm is most efficient when the length-limit is relatively relaxed.