МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2007
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2007
Proceedings of the Thirty Sixth Spring Conference of
the Union of Bulgarian Mathematicians
St. Konstantin & Elena resort, Varna, April 2–6, 2007

REPRESENTING TREES*

Boyko B. Bantchev

Trees are a well studied and widely used data structure. Nevertheless, it seems that there are topics related to trees which have not received enough attention in the computing literature. This paper discusses of them: some finer points in representing general trees.

1. Introduction. The tree data structure is the archetypal for representing hierarchies, and as such is rightly considered as a well known object in computing. Still, studying the relevant literature, as well as practicing with various programming systems reveals that some topics seem to have escaped the attention of both scientist and practitioners. These are perhaps small topics, yet we consider it important to subject them to explicit treatment. In the following, we discuss one such topic, namely the efficient, with respect to traversing and other "mass" operations, representation of general trees.

We observe that the well known ways to represent trees are either conceptually inadequate in the stated respect, or redundant, or both. Yet, it turns out to be difficult to achieve substantial improvement over them. Besides, different schemes of representing trees prove advantageous in different cases, so the goal of finding a representation "good enough" for most uses is probably unacheavable. Perhaps this is one reason why trees, despite their apparent utility, are extremely rare inhabitants of the libraries coming with the programming languages and development environments.

2. Trees and representations. Trees organize data hierarchically, but how do we represent trees as data themselves? There are many ways to do that, depending on what kind of trees are to be dealt with, the set of operations that we want to be available on those trees, and the time-space efficiency tradeoff tolerated in a particular context.

Let us concentrate on general (i.e. not binary) rooted trees. Many interesting representations of such trees are described in the classic encyclopaedia [1]. One can observe that they vary widely in terms of utility and economy of storage.

As a rule, compact representations are of lesser utility. For example, a tree of n nodes can be represented as an array of n indices, where each entry corresponds to a tree node and holds an index to that node's parent. This scheme is attractive by being both simple and the tightest possible, and it works well if all we need to do with a tree is adding a node and knowing the parent of a node. However, an instant access from a node is only given to its parent, finding any other related node, such as a child or a sibling, takes many primitive operations on the representation. As a consequence, many operations, such

*This work has been supported in part by SUFSI grant #135/2006

as removing a node, pruning or extracting a subtree, traversing, reorganizing, finding the set of children or siblings of a node, telling whether a node is a leaf etc., are very time-consuming, even prohibitively so.

We are often interested in representations that may not be as tight as the above mentioned but ensure fast access to all "close relatives" of a node – its children, its siblings and its parent, so that many useful operations on trees can be performed efficiently.

The tradition dictates that, trees being dynamic structures, each of their nodes is separately allocated and linked to some others. Depending on which links are directly provided, each of the three mentioned kinds of access appears to be direct or not.

Typically, a node is linked to one of its children and to one of its siblings. This implies that the siblings of a node form a sequentially linked list, so that starting from a node, each other one is accessible. The last node in the list can point to those nodes' common parent, thus, providing an indirect linking of all siblings to their parent. Alternatively, each node can be linked directly to its parent, so that all nodes access their respective parents instantly, but then more storage is allocated for each node and for the tree as a whole.

Trading storage for speed is acceptable in most cases, but sometimes we would need both good time efficiency and a memory usage as low as possible. Can we do better in this respect than the above sketched schemes, and if yes, how?

3. Representing a tree efficiently. First of all, vectors can be used instead of linked lists for representing children: each node's child nodes then form a vector of siblings which is referred to from that node. Thus, the sibling relation becomes implicit, and omitting the pointers from the linked representation, leads to economy of space. A certain gain in speed is also to be expected due to referencing all children of a node by index.

One does not find a mention of the vector-based representation in the well known texts on data structures, such as [2], but it is really advantageous, especially when the number of children of a node does not change frequently. Using vectors is facilitated by modern programming environments, all of which provide means for dynamic array allotting and manipulation.

Are there more sources for economising on space without sacrificing speed? Yes – although not unconditionally – and the way in which this can be done leads to an interesting observation regarding the role of hierarchy as a data structuring pattern in general.

As it was already mentioned, in representing trees – and this applies to other data structures as well – we usually rely on hierarchical composition of types. According to this view, a tree is built up of nodes, therefore, we first define the structure of a node, and then link the nodes as needed. Note that in this respect building a tree is basically the same as that of, say, a linear list, although the resulting structures are functionally different.

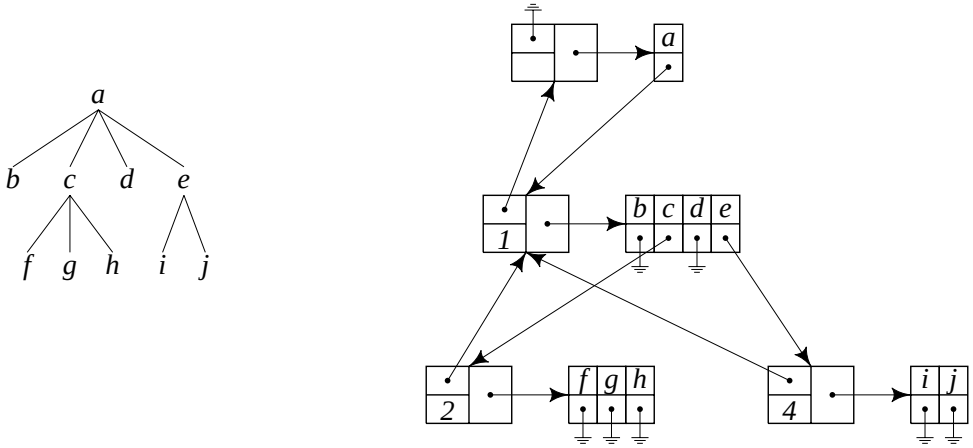
However, in this way the representation of a tree easily becomes redundant. If each node holds a reference to its parent, then, obviously all the children of a node refer to that same node but we fail to factor this out. This is insatisfactory in two ways. From a practical viewpoint, it is potentially space-inefficient. From conceptual viewpoint, it is simply inadequate (just as any other kind of redundancy is).

It turns out that representing (under our requirements) a tree – the archetypical hierarchical structure – is inadequate to do in a strictly hierarchical manner! Indeed, the

many-to-one relation between a bunch of nodes and their parent is "reverse-hierarchical" and is not readily accommodated within a straight hierarchy.

It is easy to remove the redundancy by introducing a *node handle* to mediate the connection between a node and its children. The figure shows a tree and its representation using node handles. There is a handle for each node with at least one child. Nodes themselves are grouped into blocks of siblings, each node being linked to a handle if it has children, or holding a null pointer if it is a leaf. A handle contains two pointers. One of them (heading to the right) links to a block of children nodes. The other pointer (depicted by upward arrow) links to a parent's handle and is accompanied by an index (the value below it) distinguishing the parent among its siblings. For example, the handle pointing to the block of nodes f, g, h also points to a handle pointing to b, c, d, e. The value 2 means that the second of these, i.e. c, is the parent of f, g and h. The topmost handle has a null pointer as there is no parent to link to there.

Note that this scheme readily lends itself to representing a forest. Indeed, if the topmost node vector contains two or more siblings, the structure corresponds to a forest. All usual operations, such as adding a tree (or a forest) to a forest, linking a tree as a child to another tree's node etc., are easily implementable.



In the above described representation, each handle uniquely determines a node – the parent of some group of nodes that it points to by its right (child) arrow on the figure. Since no leaf node has a handle, it may be practical to create handles for leaves, with null child pointers, temporarily as need arises, in order to identify all nodes uniformly by the same kind of data values. Those handles would not be parts of the data structure representing the tree per se, so that no space is consumed without necessity.

How much space, if any, do we save using the proposed method? We are saving some space by representing each parent uniquely, but introducing node handles increases the number of pointers used per node, so that in fact the "gain" can be negative. Therefore, roughly speaking, the more leaves and less internal nodes in a tree, the less space is consumed.

More precisely, let a tree has n nodes, m of which are internal. With a straightforward vector representation we would be using three cells for each node: for pointing to its contents, to its parent, and to the array of its children. That makes $3\,n$ pointer cells in

total. With a vector representation with handles, we need a pointer to contents and a pointer to children, plus $m+1$ handles, each comprising of three cells: $2\,n+3\,(m+1)$ cells in total. The difference between the two computed values is

$$G(n,m) = n - 3\,(m+1).$$

When G is negative, then using handles looses to the plain scheme. The worst case for G ($m = n-1$, a linear tree) amounts to $-2\,n$.

On the other hand, for a d-high complete k-ary tree $n = (k^{d+1}-1)/(k-1)$ and $m = (k^d-1)/(k-1)$, so $G = (k^d(k-3)-3k+5)/(k-1)$. A complete tree is most favourable to the handle-based representation because the number of internal nodes is the least possible. Still, as we see from the above result, for the handle-based scheme to be advantageous $k > 3$ is needed. Clearly, with k and d increasing, G grows, and is of the same order of magnitude as n, therefore, with the introduction of handles we would be using about $2/3$ of the space that the simple vector scheme needs.

3. Conclusion. What can we conclude from our observations? For small or medium-size trees, or ones with low degree of branching, it is best to use the simple vector representation. For large trees the handle-based representation consumes less space. The difference between the two is insignificant for many applications, but there remain cases when it is important. Huge XML files – something not rare in today's information technology world – are an example.

REFERENCES

[1] D. E. Knuth. The art of computer programming, Vol. 1: Fundamental algorithms, 3rd ed. Addison-Wesley, 1997.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman. Data structures and algorithms. Addison-Wesley, 1983.

Boyko B. Bantchev
Institute of Mathematics and Informatics
Acad. G. Bontchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: bantchev@math.bas.bg

ПРЕДСТАВЯНЕ НА ДЪРВЕТА

Бойко Бл. Банчев

Дърветата са добре изучена и широко използвана структура от данни. При все това, някои свързани с дървета теми изглежда не получават заслужено внимание в литературата от съответната област. Тази статия обсъжда една такава тема: някои подробности във връзка с представяне на общи дървета.