Differential binary trees

A fast and compact data structure.

By Adriaan van Os <<adriaan@adriaan.biz>>

Differential binary trees store data embedded in the flow of the tree. This results in low data redundancy and fast execution of indexed operations, bridging the gap between arrays and binary trees.

Contents

1. Classical binary trees

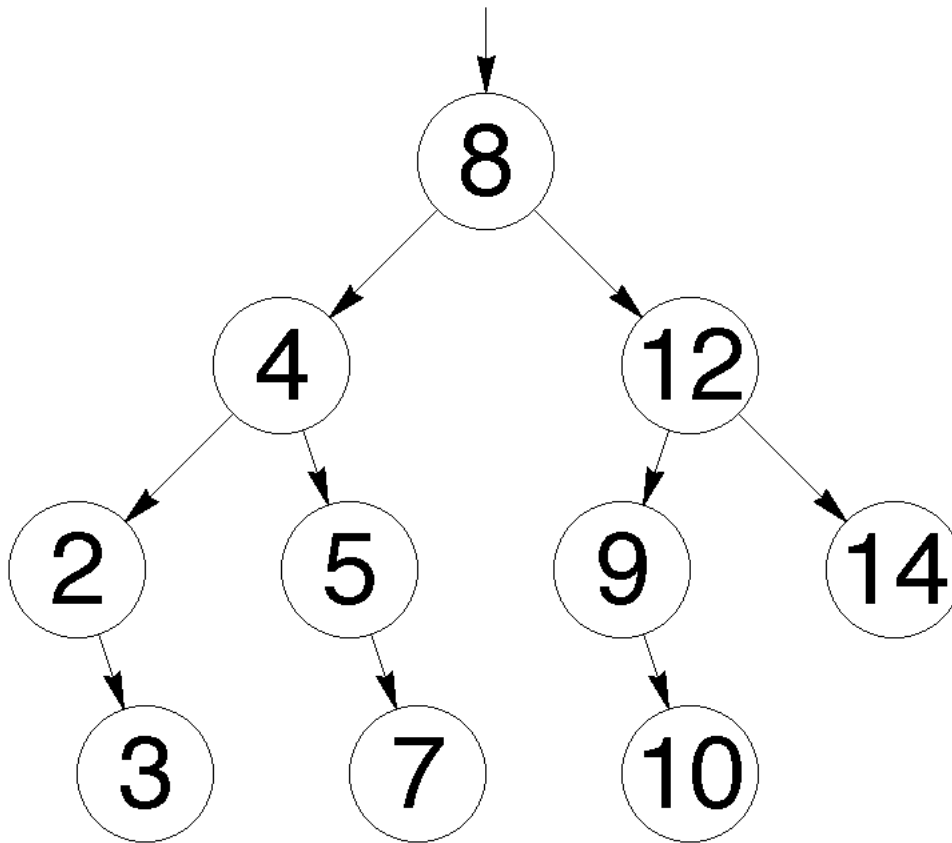Consider the binary tree of figure 1 (below).

Figure 1. A binary tree with absolute key values

This is a common data structure. Still, if we look carefully at the tree, the following seems odd:

> *the key values are **isolated**, they are not part of the **flow** of the tree*

This suggests that representation of data keys in the tree:

• forgets to use the fact that the tree is *ordered*
• is through repetition *partly redundant*
• is *unnatural* and therefore *inefficient* for certain algorithms that operate on the tree.

With *unnatural* I mean that something is wrong (wringing), namely:

• to reach a data key, we proceed along a path in the tree, we gradually reach the point where the it is, but
• if we reach that point, the data key itself is suddenly there, its value *did not also grow* along the path.

If the key values were weights in a natural tree, it would fall to the right.

We will look at the consequences more closely in the rest of this thesis.

2. Differential binary trees

So, let's have a look at a tree where the keys themselves do grow along the path that we follow to find them. Such a tree is given in figure 2 (below). It represents the same data as the classical binary tree of figure 1 (above).
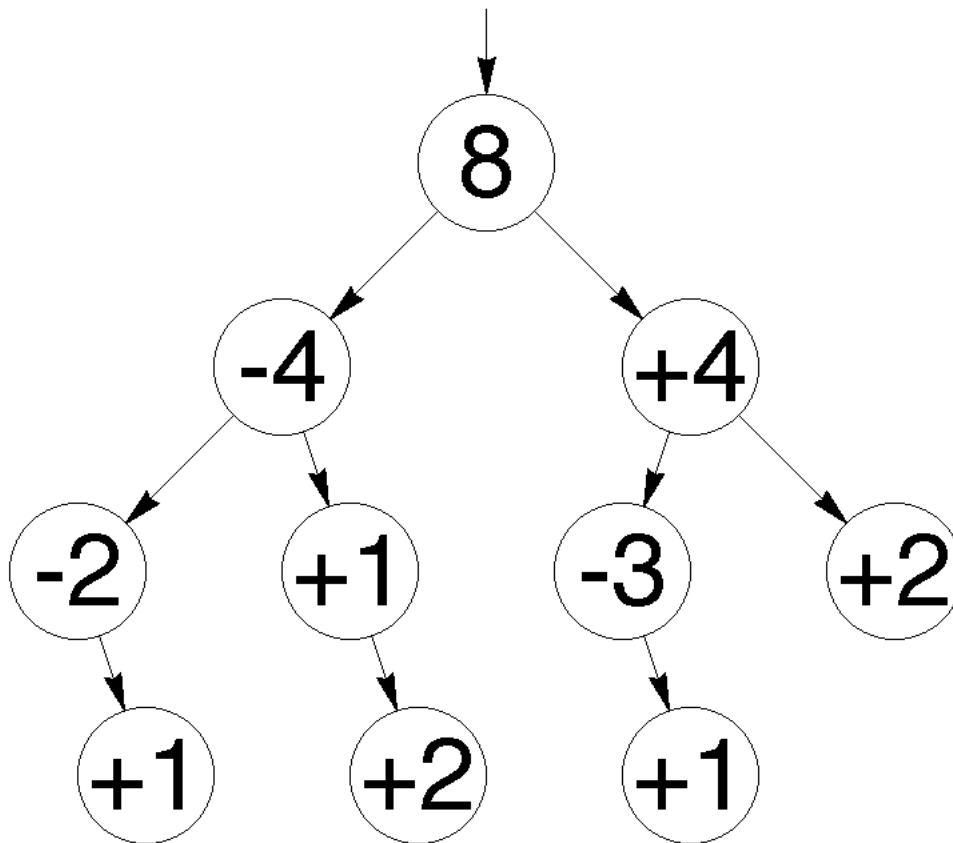

Figure 2. A binary tree with differential key values embedded in the flow of the tree

The tree of figure 2 (above) stores keys as follows:

• the root node stores its own key value
• each successive node stores the difference between its own key and its parent's key.

So, for example node 4 stores 4 - 8 = -4 and node 5 stores 5 - 4 = 5 - (8 - 4) = +1. Therefore we will call the tree a *differential tree*.

The following is true for differential binary trees:

• the *sum* of all stored node key differences (diffs) along the path to a node, produces the node's key
• the left child node of any node has a *negative* node diff.
• the right child node of any node has a *positive* node diff
• the absolute value of node diffs tends to decrease with the depth of the node.

The latter observation is interesting. Each subtree cuts the number of remaining nodes into half. In the special case that the tree is balanced and completely filled, each subtree level decreases the number of bits required for key diff representation with exactly one. See figure 3 and 4 (below).

Note that there are twice as much nodes that require n bits as there are nodes that require n + 1 bits !
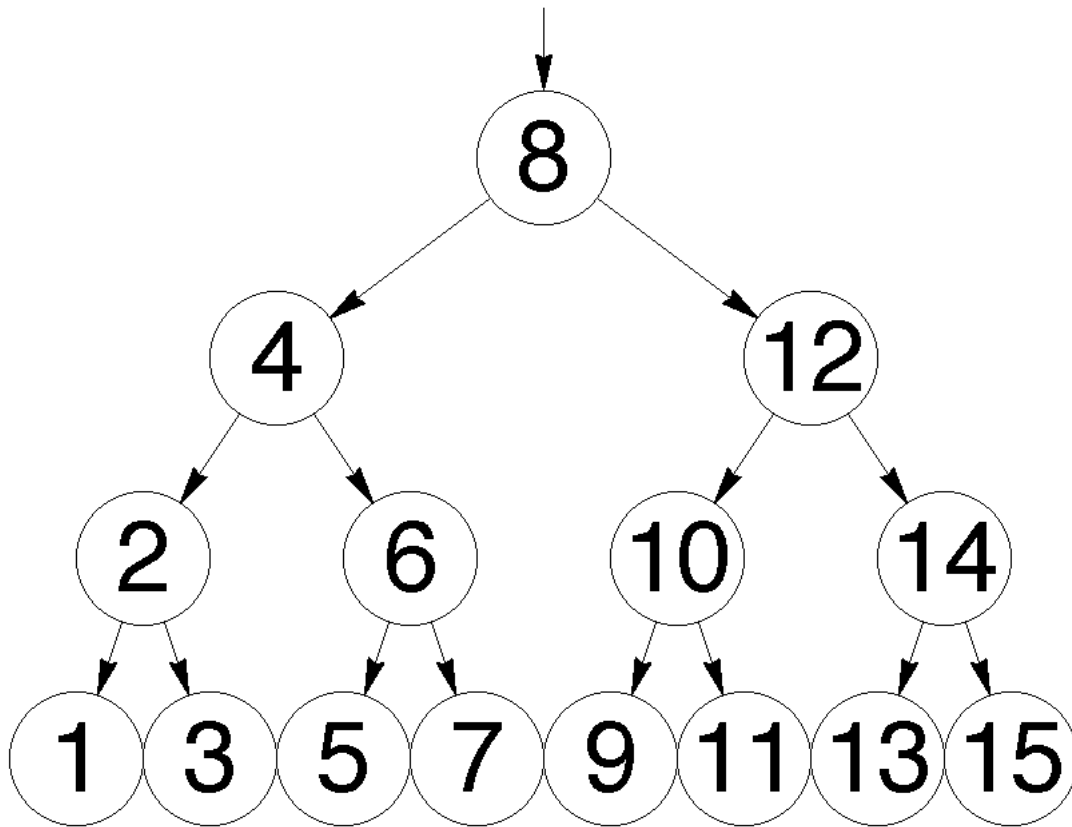
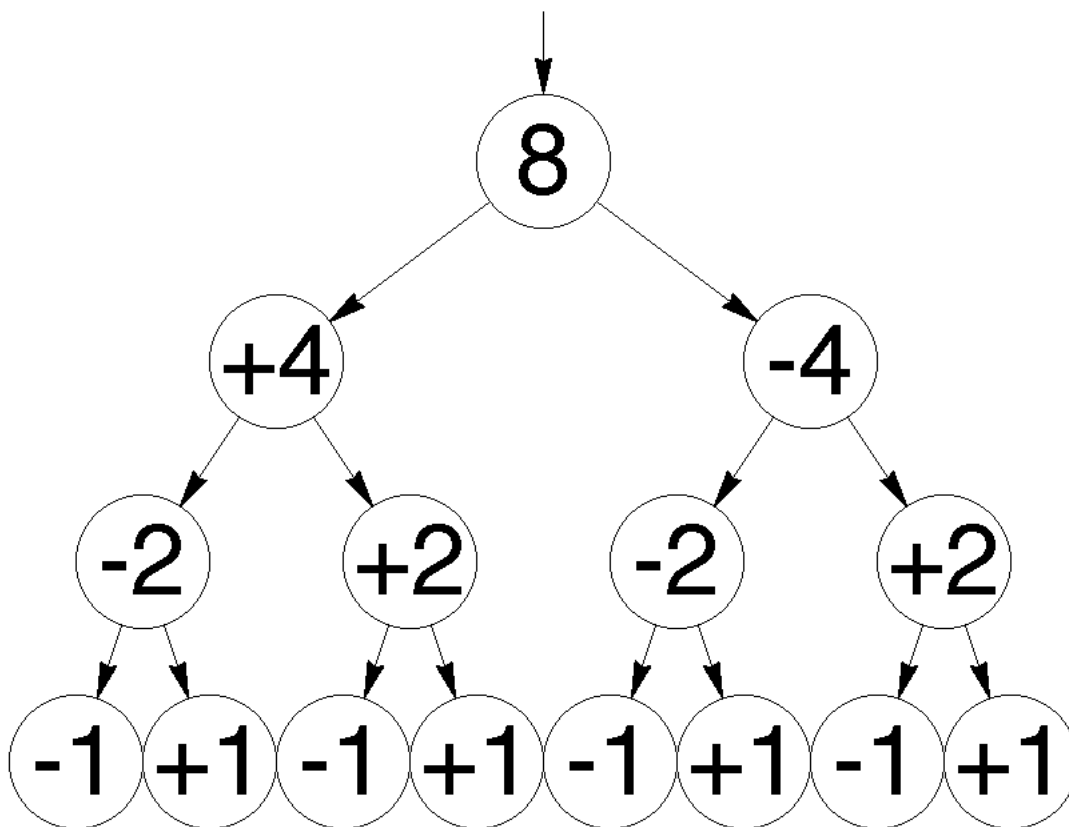*Figure 3*. An ideally filled and balanced binary tree with absolute key values

Figure 4. An ideally filled and balanced binary tree with differential key values

We could say that the data 'compression' of the differential binary tree is at its best when the tree is ideally filled and balanced. However, from the perspective of the differential binary tree, there is no data compression at all, just data *redundancy* in the classical binary tree. And that data redundancy is at its peak when the tree is ideally filled and balanced.

The data efficiency of the differential binary tree depends, quite naturally, on the **sparseness** of its data keys.

Note that, if we put the differential tree of figure 4 upside-down, it resembles a real tree, with the absolute value of the node values indicating the thickness of its branches. Therefore, I call the differential tree more *natural* than the classical binary tree with absolute key values.

3. Classical arrays

Before continuing with our analysis of the differential binary tree, we will have a closer look at the classical array, implemented as a consecutive block of data in memory (figure 5 and 6) or as a consecutive block of pointers to data in memory (figure 7).
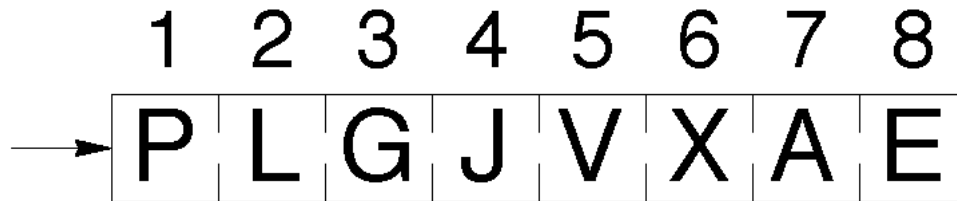

Figure 5. An unordered array, implemented as a consecutive block of data in memory
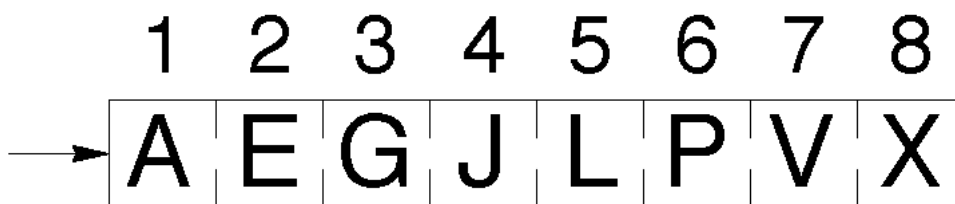
Figure 6. An ordered array, implemented as a consecutive block of data in memory

Figure 7. An ordered array, implemented as a consecutive block of pointers to data in memory

In an array, data is *indexed*. This means that
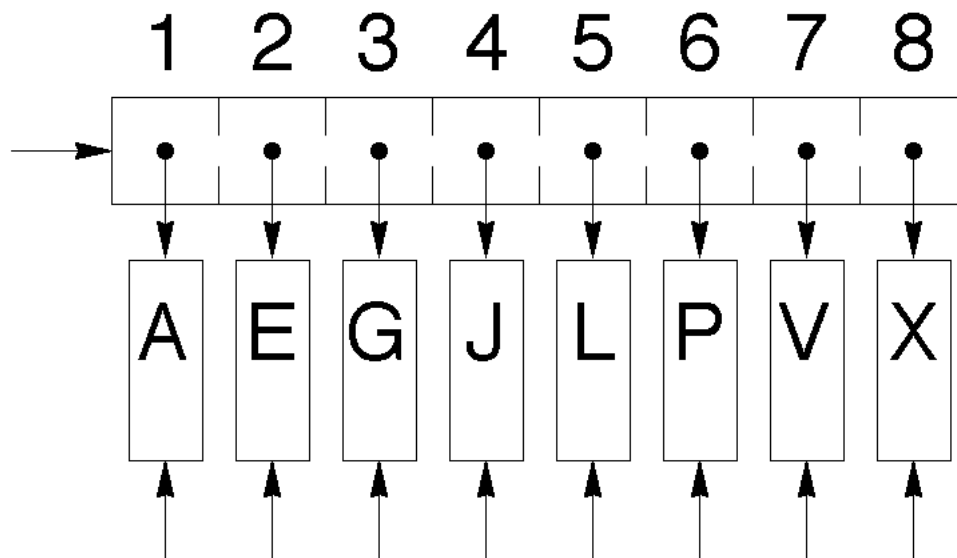
• programming is easy
• there is no need for a key inherent to the data (in other words, the index is neutral to and separate from the array's data)
• the array is a natural choice if the problem is to store an indexed list of data.

	unordered array of consecutive data	ordered array of consecutive data	ordered array of consecutive pointers to data
indexed access	1	1	1
find	N	$^2\log(N)$	$^2\log(N)$
indexed append	1 (plus realloc delay)	1 (plus realloc delay)	1 (plus realloc delay)
indexed insert	N (plus realloc delay)	N (plus realloc delay)	N (plus realloc delay)
indexed delete	N	N	N
iteration over array	N	N	N
delete all	1	1	N
individual array elements can be referenced	no (physical pointers are not persistent)	no (physical pointers are not persistent)	yes (physical pointers are persistent)
can have variable-size elements	no	no	yes
data overhead per element	none	none	pointer-size bytes
array can be sparse	no	no	will still cost pointer-size bytes per element for each nil pointer

Table 1. Properties and approximate order of needed time for operations in arrays, where N is the number of elements in the array.

Table 1 (above) lists the relative performance of arrays for basic operations. We can improve the key-find time from order N to order $^2\log(N)$ using *binary search* in an ordered array. Still, the general conclusion for classical arrays can only be that they are:

• fast and compact for static data
° slow for dynamic data, unless the number of elements N is quite small.

4. Binary trees compared

Table 2 (below) lists the properties and performance of operations in classical binary trees, compared to differential binary trees, assuming that both are implemented as a near-balanced tree, for example as AVL tree or as red-black tree.

	classical binary tree, implemented as AVL or red-black tree	differential binary tree, implemented as AVL or red-black tree
access	$^2\log(N)$	$^2\log(N)$
indexed access	$^2\log(N)$	$^2\log(N)$
find	$^2\log(N)$	$^2\log(N)$
append	$^2\log(N)$	$^2\log(N)$
indexed append	$^2\log(N)$	$^2\log(N)$
insert	$^2\log(N)$	$^2\log(N)$
indexed insert	N	$^2\log(N)$
delete	$^2\log(N)$	$^2\log(N)$
indexed delete	N	$^2\log(N)$
key shift	N	$^2\log(N)$
iteration over tree	N	N
delete all	N	N
individual tree elements can be referenced	yes (physical pointers are persistent)	yes (physical pointers are persistent)
can have variable-size elements	yes	yes
data overhead per element	2 * pointer-size bytes plus 1 or 2 balancing bits	2 * pointer-size bytes plus 1 or 2 balancing bits
tree can be sparse	yes	yes

Table 2. Properties and approximate order of needed time for operations in binary trees, where N is the number of elements in the array.

The *indexed* operations mimic the same operations in an array. This is important if we want to implement an array using a tree. I recall the bad performance of classical arrays, outlined in the chapter above.

So, for example, if we insert an element in an array at index 8, we first have to move all data at position 8 and above, one position up. Doing the same in a tree that implements an array (with index numbers as keys), we have to add one to all index keys from 8 upwards.

This very **index key shift** is a slow (order N) operation in a classical binary tree, but a fast (order $^2\log(N)$) operation in a differential binary tree ! Right here we see the advantage of storing keys differentially, that is, integrated into the natural flow of the tree.

5. Bridging arrays and trees

From the above, it may be clear that the differential binary tree bridges the gap between arrays and binary trees, by making *index key shift* a cheap operation.

In fact, the classical array can be looked at as a differential data structure with an *implicit index key difference* of one, based on the fact that the data structure is *non-sparse.* I recall the resemblance with figure 4 (above).

Using differential binary trees, we can implement dynamic arrays so that

- all indexed operations are guaranteed of order $^2\log(N)$
- iteration is of order N
- the array can be sparse.

Note that iteration is a more efficient operation than indexed access in a loop.

Table 3 summarizes the properties of array implementations we have discussed.

	unordered array of consecutive data	ordered attay of consecutive data	ordered array of consecutive pointers to data	array implemented with classical AVL or red-black tree	array implemented with differential AVL or red-black tree
indexed access	1	1	1	$^2\log(N)$	$^2\log(N)$
find	N	$^2\log(N)$	$^2\log(N)$	$^2\log(N)$	$^2\log(N)$
indexed append	1 (plus realloc delay)	1 (plus realloc delay)	1 (plus realloc delay)	$^2\log(N)$	$^2\log(N)$
indexed insert	N (plus realloc delay)	N (plus realloc delay)	N (plus realloc delay)	N	$^2\log(N)$
indexed delete	N	N	N	N	$^2\log(N)$
iteration over array	N	N	N	N	N
delete all	1	1	N	N	N
individual array elements can be referenced	no (physical pointers are not persistent)	no (physical pointers are not persistent)	yes	yes	yes
can have variable-size elements	no	no	yes	yes	yes
data overhead per element	none	none	pointer-size bytes	2 * pointer-size bytes plus 1 or 2 balancing bits	2 * pointer-size bytes plus 1 or 2 balancing bits
array can be	no	no	will still cost	yes	yes

sparse			pointer-size bytes per element for each nil pointer		

Table 3. Properties and approximate order of needed time for operations in *array implementations, where N is the number of elements in the array.*

6. Algorithms

A detailed discussion of all aspects of the implementation of differential binary trees extends the scope of this publication. Let me just issue a few remarks.

• while searching for a key, we ***change*** the key to look for in a subtree, e.g. if we look for key 5 in the tree of figure 1 and 2 (above), we subsequently look for keys -3 and +1 in a subtree; we stop when we have found a difference of 0 or a leaf node.

• operations like *insertion, deletion* and AVL-*rotation* require diff-key adaption of nodes involved

• the several different *iteration* algorithms (forward, backward, from, until, deleting, threaded) are worth studying.