Remapping a Sorted Array to a Complete Binary Tree / Breath-First Ordering

David M. Dantowitz, david@dantowitz.com
May 1, 2018

Overview

An efficient O(N) algorithm is presented to remap a static sorted array to breadth-first order. This remapping preserves the sort property such that the resulting array may be traversed as a complete binary tree with implied bidirectional links.

The key concept of the algorithm is the recursive partitioning of a sorted array such that there exists a remapping of at least one partition to a full binary tree. When the array is completely partitioned and remapped, the result is a complete binary tree. Additionally, two optimizations of the algorithm are presented, resulting in a 5.3 times increase in performance.

Background

Finding an element in a contiguous sorted array is easily accomplished via Binary Search. The key operation at each step being the selection of the midpoint of a range:

```
midPoint =(high+low)/2
```

or if overflow is a concern:

```
midPoint=low+(high-low)/2
```

(Divide by 2, can of course be replaced by a right bit shift.)

Thus, with a sorted array, the right and left branching of the implied binary tree is built into the ordering of the array.

Seeking to further improve performance and usefulness, we remap the sorted array into an array that reflects the ordering of a complete binary tree (CBT) or breadth-first ordering. This has several benefits:

1) Computation the root of sub trees or the midpoint of a sequence is simpler. For the first entry in the array:

   midPoint=0 (the first element of the array)

   Assuming an origin of 0, subsequent sub tree roots or midpoints of sub ranges are:

   (midpoint+1)*2-1 (left branch) and (midpoint+1)*2 (right branch)

   (or midPoint+midPoint, if you'd rather avoid using a multiplication). This method removes the divide by two or bit shift right when partitioning a range.

2) Akin to a sorted array, the need to store pointers is removed in exchange for a computation to reach the next midpoint. A subsequent advantage of using an array with CBT ordering is the implied pointer to the parent of an entry in the array, were it in a binary tree:

    parentIndex = (nodeIndex-1)/2

    Thus you have all the benefits of a complete (balanced) binary tree with both child and parent links at no cost of storage.

There may be other advantages, but for my intended use these were sufficient motivation to seek an efficient method to remap an ordered array into a CBT ordering.

A simple example would be a sorted list of 7 elements: 1 2 3 4 5 6 7

Remapping them to a CBT order would yield: 4 2 6 1 3 5 7, recognizable as a breadth-first traversal of a balanced binary tree based on the integers [1..7].

Examining the Data: First to the Forest

Given a set of N integers in a balanced binary tree, they can be viewed as a root node with two smaller trees:

Root
Left Tree Right Tree
Figure 1

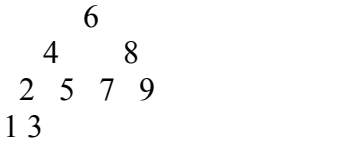
If the tree is a full and balanced binary tree, then $N = 2^a-1$ and both the left and right trees contain $2^{a-1}-1$ nodes (i.e., the root plus two sub trees is $2*(2^{a-1}-1)+1= 2^a-1$ nodes.) Thus if the original tree is full, the sub-trees are also full.

If the tree is not full, then three cases exist: the left tree is full and the right tree is not, vice versa and third, left and right are full, but left's height is the right's height+1.

To remap a full binary tree in a sorted array to an array with CBT ordering is simple: partition the array into left and right sections by selecting the midpoint, place the midpoint into the CBT array, then perform the same operation on the two ranges (representing the two sub trees), placing each root (midpoint) left, then right into the CBT array.

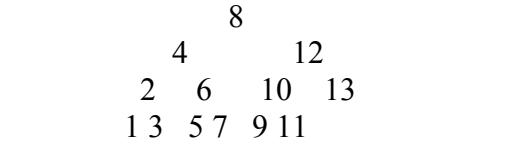
Things get far more interesting when you need to select a partition for a tree that is not full.

Given a non-full binary tree (a list of N integers where $N \neq 2^a-1$ for any a), you need to find a partition point that always gives you a full, balanced binary tree on at least one side. The trees in figures 2 and 3 are examples of the right and the left sub trees being full, respectively:



Right sub tree is full: nodes 7-9

Figure 2



Left sub tree is full: nodes 1-7

Figure 3

Figure 2's left sub tree (contains 1 through 5) is not full, but the right sub tree 4 is full (7 through 9). Figure 3's left sub tree (1 through 7) is full, but the right sub tree (9 through 13) is not full.

You will notice that when the left sub tree is full, the number of leaf nodes in the bottom row is greater than or equal to half the max number of leaves possible. If the left sub tree is not full then the bottom row will contain fewer than half the number of possible leaves. So, next we devise a method for determining the proper partition of a sub array, such that at least one of the sub ranges or sub trees contain a full tree.

To determine how many leaf nodes will fit in the bottom row of a tree, we take the height of the tree H (root to the left-most node) and compute the max leaves that would fit in the bottom row $M=2^{H-1}$. The height, H, is $\text{trunc}(\log_2(N))+1$, where N is the total number of nodes in the tree, or the length of the sorted array.

Thus, we have

N=number of nodes (length of the sorted array or current sub array)
H=height of the tree that would hold those nodes (root to the left-most node)
M=max possible number of leaves in the bottom row of a tree of height H

A full tree of height H will have 2^H-1 nodes, and since $M=2^{H-1}$, M is both the maximum number of leaves in the last row and one more than the total number of nodes in rows 1 to H-1 (the tree above the final row, which is always a full tree).

Based on N, the actual number of nodes in the tree and M-1, the number of nodes in the tree if we dropped the last row (because a complete binary tree of height H is always full in levels 1 to H-1), we may compute the actual number of leaves in the bottom row of the tree as:

$B = N-(M-1)$ *or* $B = N-M+1$

Next, to see if the number of leaves in the final row, B, is at least half of M, we test:

$B \geq M/2$, because M/2, (half the row size) is important, assign $Z=M/2$

$B \geq Z$, substitute for B (note that $M=2Z$, so $B=N-2Z+1$)

$N-2Z+1 \geq Z$, Collect the Zs and the constant 1 on the right

$N \geq 3Z-1$, which leaves N on the left

Using our definitions above, we can compute Z, half the max length of the bottom-most row in the tree containing N nodes as:

$$Z = M/2$$
$$Z = (2^{H-1})/2$$
$$Z = (2^{trunc(log2(N))+1-1})/2$$
$$Z = (2^{trunc(log2(N))})/2$$
$$Z = 2^{trunc(log2(N))-1}$$

Note that Z is also 1+ (the number of nodes in a full tree of height H-2).

Now, to determine where to partition a sorted array into a root and two sub trees, where at least one is a full tree, we do the following:

If ($N \geq 3Z-1$)

partition off (Z-1)+Z nodes from the start of the range, for a full left sub tree. As seen in Figure 3, the bottom row is at least half full, so the left sub tree is full. Removing the root (the partition value) the nodes placed in the left sub tree begin at low and include all values up to low+(Z-1+Z)-1, which 2^{H-1}-1 nodes. The remaining nodes greater than the selected root form the right sub tree.

otherwise

partition off Z-1 nodes from the end of the range, for a full right sub tree and note that the height of the sub tree will be two less than the height of the original (as we recall that Z-1 is equal to 2^{H-2}-1, the number of nodes in a full tree of height H-2).

Out of the Forest and into the Code

Here we work with two arrays of N integers. The first is sorted, the second will be the result from remapping the sorted array into CBT order or Breadth-First order.

sortedArray is an array of count integers in ascending order
reMapped is an array of count integers, remapped from the sortedArray in CBT order

```
Call:  reOrderTree1(sortedArray, reMapped, 1, count, 1);

void reOrderTree1(int *oldArray, int *newArray, int low, int high, int base)
{
       int N, root, Z;

       if (low == high)                                // root case
              {
              newArray[base-1] = oldArray[low-1];     // save (single) element
              return;
              }

       N = high-low+1;     // number of entries in this range

// Find the new root / partition point
       Z = pow(2, trunc(log2(N))-1);

       if (N+1<3*Z)
              root=N+1-Z;         // not at or past the half way mark on last row
       else
              root=Z+Z;

       newArray[base-1] = oldArray[root+low-2];       // save partition element

// We always have a left subtree. Sometimes have a right subtree
       reOrderTree1(oldArray, newArray, low, root+low-2, base*2);
       if (N>2)
              reOrderTree1(oldArray, newArray, root+low, high, base*2+1);

       return;
}
```

If we optimize a bit and move the pow(log2) call outside the routine and instead reduce Z by a factor of 2 for each partitioning, we achieve a 3.7 time speed up when remapping an array of the integers 1 to 100 million for ten iterations.

```
Call:  Z = pow(2, trunc(log2(count))-1);
       reOrderTree1a(sortedArray, reMapped, 1, count, 1, Z);

void reOrderTree1a(int *oldArray, int *newArray, int low, int high, int base, int Z)
{
       int N, root;

       if (low == high)                                  // root case
              {
              newArray[base-1] = oldArray[low-1];      // save (single) element
              return;
              }

       N = high-low+1;            // number of entries in this range

// Find the new root / partition point
       if (N+1<3*Z)
              root=N+1-Z;         // not at or past the half way mark on last row
       else
              root=Z+Z;

       newArray[base-1] = oldArray[root+low-2];       // save partition element

       Z = Z>>1;                  // reduce by a factor of 2

// We always have a left subtree. Sometimes have a right subtree
       reOrderTree1a(oldArray, newArray, low, root+low-2, base*2, Z);
       if (N>2)
              reOrderTree1a(oldArray, newArray, root+low, high, base*2+1, Z);

       return;
}
```

We can optimize further by changing the base case from a sub array of 1 element to a sub array of fewer than 4 elements. Here we achieve 5.3 time speed up over the original.

```
Call:  Z = pow(2, trunc(log2(count))-1);
       reOrderTree1b(sortedArray, reMapped, 1, count, 1, Z);

void reOrderTree1b(int *oldArray, int *newArray, int low, int high, int base, int Z)
{
       int N, root;

       N = high-low+1;                  // number of entries in this range

// Optimize small sub trees. If N is 1, 2, or 3, we know the order, so no need to
// make a recursive call, these three sizes are our base case.

       if (N < 4)
              {
              if (1==N)
                     newArray[base-1] = oldArray[low-1];     // single element
              else
                     {
                     newArray[base-1] = oldArray[low];       // second element
                     newArray[base*2-1] = oldArray[low-1];   // left branch
                     if (3 == N)
                            newArray[base*2] = oldArray[low+1];     // right branch
                     }
              return;
              }

// Find the new root (partition point)
       if (N+1<3*Z)
              root=N+1-Z;  // not at or past the half way mark on last row
       else
              root=Z+Z;

       newArray[base-1] = oldArray[root+low-2];       // save partition element

       Z = Z>>1;           // reduce by a factor of 2

       reOrderTree1b(oldArray, newArray, low, root+low-2, base*2, Z);
       reOrderTree1b(oldArray, newArray, root+low, high, base*2+1, Z);
}
```

Conclusions

This algorithm may find use where data is static over a period of time, used in search, and storage space is at a premium: embedded devices, pre-computed data, tables, or patterns. Storing a complete binary tree in a linear array results in a compact representation of the tree's data with implied bidirectional pointers between parent and child nodes (e.g., they are computed, thus require no storage).

Finally, I'll note that although removing recursion from the solution did not achieve appreciable speed up, the algorithm is easily adapted to run in parallel.