Direct Construction of a Complete Binary Search Tree

Emmet Caulfield

June 9, 2016

Abstract

A complete binary search tree (CBST) of the integers $\{1, 2, \ldots, n\}$ is a) unique and b) defined completely by the size, n.

Accordingly, if data size is known *a priori*, a complete binary search tree (CBST) can be constructed in $O(n \log n)$ time from any in-order data (in a file or array) with neglibible memory overhead. This is achieved by placing the ordered elements directly at the correct indices in an array such that "left" and "right" index calculations, most often associated with *heaps*, can be used in lieu of pointers, yielding $\Theta(n)$ memory savings over the more conventional dynamic node based equivalent.

This method of CBST construction is computationally cheap, using only integer comparisons irrespective of the data; this is advantageous where actual data comparisons are expensive.

In addition, the conventional construction method for a CBST always constructs a pathological tree from sorted input, which is not a problem for this method.

1 Motivation

Consider a 3D cartesian simulation grid with, say, 512 cells on a side[1] represented by an array of cell property structures (or parallel property arrays) and an array of pairs of cell IDs representing the faces (a "connection list"). Cell IDs may be assigned monotonically, but not strictly sequentially (i.e. there are gaps), either for legacy reasons or because of some embedded semantic.

For certain "pre-simulation" manipulations, utility software sometimes uses a binary search tree of cell IDs. This is typically constructed in the usual way, having a `tree_node` (or similar) structure with left and right pointers in addition to cell data.

In a classic binary heap structure, left and right pointers are unecessary, their place being taken by left and right index computations. If a binary search tree for the 512^3 grid is represented in this way, the memory saving, in eliminating 2 64-bit pointers per node, is 2GiB ($2 \times 512^3 \times 8$).

The asymptotic search cost, of course, remains $O(\log n)$, although it is unknown whether the poor, if well-understood, cache behavior of the heap operations used here is, on average, better or worse than the pointer-chasing equivalents.

[1] This figure is loosely based on a 200 Mcell requirement for a certain toolkit.

2 Description

2.1 Overview

The key to this method is the observation that, given input data of size n, the value at the root of a complete binary search tree (CBST) always comes from the same position in the input. Moreover, owing to the uniqueness theorem for CBSTs, the map from input array positions to CBST positions is bijective, given input size. Therefore, provided that the input is ordered, the position in the input of the element at the root of the CBST can be computed arithmetically. Once this value is established, the remaining indices can be calculated recursively or by equivalent iteration.

2.2 Detail

The height, h, of a CBST of size n is given by $h = \lceil \log_2(n) \rceil$. The smallest input element (at position 0 in the input) always goes in the "bottom left corner" at CBST index i_0, where $i_0 = 2^{h-1} - 1$. In the heap representation, this is, by definition, also the number of elements before the smallest element in the CBST. The number of elements in the last row, or width, w, is then given by $w = n - i_0$.

Consider the CBST of size $n = 10$. It has height $h = 4$. The smallest input element, from position 0 in the ordered input, always goes in (zero based) position $i_0 = 7$ in the CBST. The number of elements in the last row is then $w = n - i_0 = 10 - 7 = 3$.

It is convenient to define the last row's half capacity, k, as $k = 2^{h-2}$. For the CBST of size $n = 10$ discussed above, $k = 4$.

If the left half-tree is *full* ($w \geq k$), the"value" in the CBST root, i.e. the (zero based) index in the input of the value that ends up in the root of the CBST, is *also* i_0.

If the left half-tree is *not* full ($w < k$), on the other hand, the "value" in the root is less than i_0 by $k - w$, i.e. exactly the number of elements that it would take to fill the empty right side of the last row.

In general, then, combining these two cases, the input index of the value in the root of the CBST is given by $i_0 - \max(k - w, 0)$.

3 Example

We need the conventional heap *left* and *right* index operations and the height[2]:

```
int left  (int i) { return (i<<1)+1; }
int right (int i) { return (i<<1)+2; }
int height(int n) {
    int h=0;
    while(n) {
        n >>= 1;
```

[2]The height is more efficiently, if less portably, computed using a `CLZ` (count leading zeros) instruction, available on most hardware. In code intended for compilation with `gcc`, the height function reduces to `return 8*sizeof(int)-__builtin_clz(n);`

```
        h++;
    }
    return h;
}
```

The "half capacity", k, can be regarded as: half the capacity of the bottom level; the capacity of the second-last level; or one more than the capacity of the left or right halves, excluding the root.

```
int cbst_root(int n)
{
    // Height of CBST of size n:
    int h = height(n);

    // CBST index of smallest value:
    int i=(1<<(h-1))-1;

    // Occupied width of the bottom level
    int w=n-i;

    // Half-capacity of bottom level:
    int k  = 1<<(h-2);

    // CBST index of the root:
    return i - MAX(k-w,0);
}
```

Once the CBST index of the root element is available, computing the CBST index for any input position, p, in data of size n is most easily achieved by "simulation", i.e. as if we were building the CBST:

```
size_t cbst_index(size_t n, size_t p)
{
    size_t index=0;
    size_t root=1;

    while( root ) {
        root = cbst_root(n);
        if( p > root ) {
            index  = right(index);
            n -= root + 1;
            p -= root + 1;
        } else if( p < root ) {
            index = left(index);
            n = root;
        } else {
            break;
        }
    }
    return index;
}
```

4 Improvements

The practical value of the improvements, suggested below, is debatable, since the time spent in `cbst_index()` is highly unlikely to be a limiting factor in any software, and it may be preferable to retain the extreme simplicity of `cbst_index()` as shown above.

4.1 Node Properties

If n is odd, the root and all internal nodes are also odd and all of the leaf nodes are even.

If n is even, the root is even, the internal nodes in the right half-tree are even, the leaf nodes of the left half-tree are even, and the leaf nodes of the right half-tree are odd.

These properties can be exploited to reduce the cost of `cbst_index()`.

For example, if n is odd, and the input position p is even, then p is in the final row and would require $\log n$ iterations of the `while` loop in `cbst_index()` (above). But, exploiting the first property observed above, p is simply at $i_0 + \frac{p}{2}$. This is also true up to $p = 2w - 2$ for even n.

In other words, an odd/even test on p and/or n enables us to short-cut $\log n$ iterations of the `while` loop in `cbst_index()` with a simple arithmetic calculation in the most expensive half of all cases.

4.2 Input Positions as Encodings of Node Positions

In the huge majority of cases, the bits of the input position p can be viewed as an encoding of the corresponding index.

Specifically, if the binary value of p is truncated to the number of bits needed to represent n, then beginning with the MSB, a value of zero corresponds to "going left" and a value of 1 corresponds to "going right". If we iterate through the bits of p in this way, performing the left and right operations on the root index, 0, we arrive at the index value for a *perfect* binary tree. Since the complete binary tree is "mostly perfect", this property can, again, be used as a short-cut to the correct value in `cbst_index()`.

Rather than merely iterating through the bits of p, counting runs of zeros or ones enables further optimization, albeit of doubtful real value.

5 Conclusion

A simple and clear method for constructing a complete binary search tree (CBST) from sorted input was presented.

A number of minor potential optimizations, exploiting CBST properties, were suggested.

Remarkably, this seemingly simple and obvious method appears nowhere in the literature[3].

[3] Or, at least, nowhere I could find.