ECE-250 – Algorithms and Data Structures (Winter 2012)

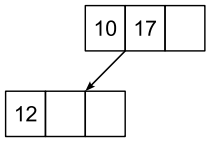
Tutorial 6 (2012-03-08) – Suggested solutions

1 – Insert the following values into an initially empty 4-way tree: 10, 17, 12, 5, 21, 43, 40, 35, 30

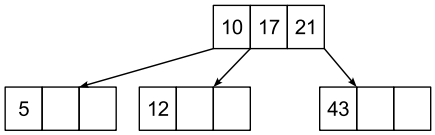
Solution:

The first insertion creates the root node, with 10 followed by two empty positions. Second insertion clearly fits at the right of 10, in the same root node.

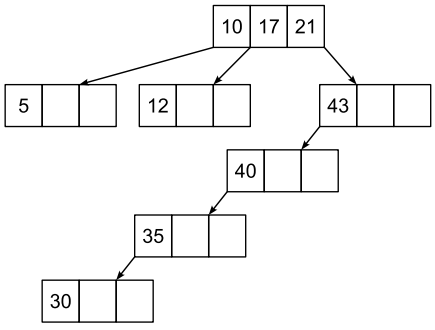
For the third insertion, there is no room at the root node; the sub-tree should be between the values 10 and 17, so it would be the second sub-tree that we create, as shown below: Then, 5 goes at the left



of 10, so we need to create that sub-tree; then, 21 can go at the root node, at the right of 17; 43 goes at the right of 21, requiring that sub-tree to be created. So far, the tree would be as shown below: The



last two insertions have quite unfortunate outcome (in terms of imbalance in the tree) — 40 goes at the left of 43 (from the root, we follow the right-most sub-tree: $40 > 21$, then $40 < 43$), so a sub-tree has to be created. Then the same happens with 35 and 30, ending up in the following tree:

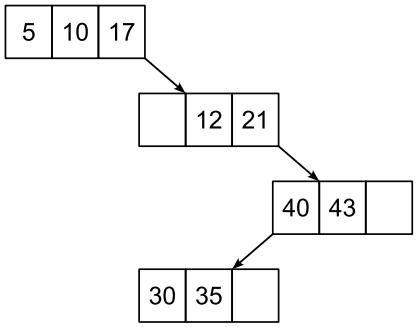


2 – Without attempting any actual balancing, is there a better strategy to place values in newly-created nodes to reduce the height of the tree, considering the average case, with random values evenly distributed? (retry the insertions, in the given order, with this new strategy)

Solution:

The situation with the last few insertions suggests that for random data, where on average we'll get subsequent values at the left and at the right of a given value, we could improve things by creating the nodes such that the first element is centered — that way, we guarantee that we have room for at least one more element (more, if it was an M-way tree with a larger M) before we need to create a sub-tree.

The resulting tree would be as follows (yes, I know, from the point of view of the root, it looks even less balanced — but the height was reduced; we notice that the nodes are highly populated by comparison, etc.):



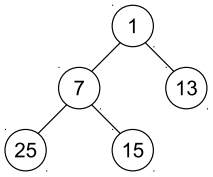
3 – (a) Insert the following values into an initially empty Heap: 7, 15, 13, 25, 1, 5, 9, ensuring that we maintain a complete binary tree.

(b) Show the procedure to remove them in the appropriate order, without worrying about maintaining a complete tree.

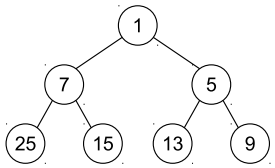
(c) Repeat, maintaining a complete binary tree (for the first two removals, use the less efficient method, and for the rest use the preferred, more efficient method)

Solution:

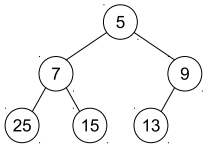
(a) To maintain a complete binary tree, we insert always at the left node following a "breadth-first traversal" pattern. 7, 15, and 13 are inserted in that order without any adjustments needed. 25 is then inserted below (left of) 15, again without any adjustments required. Then, 1 is inserted below (right of) 15, requiring a swap with 15, then requiring a swap with the root, ending up in the following: 5



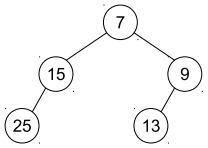
is then inserted below (left of) 13, requiring a swap with 13, and no more swaps ($1 < 5$); finally, 9 is inserted below (right of) 5, requiring no swaps. The heap's final contents is as follows:



(b) Removing 1 causes 5 (the lower of the children) to be promoted, and that one causes 9 to be promoted:



Then, removing 5 causes 7 to move up, then 15:

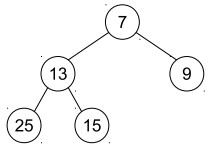


Removing 7 moves 9 up, then 13. Then removing 9 causes 13 to move to the root, and the rest goes "in straight line".

(c) The less efficient method refers to first remove without paying attention to balance/completeness of the tree, then take the element that we should have removed and patch the hole left by the removal (adjusting as needed, as we do with an insertion).

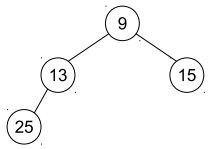
For the first removal, the hole left was precisely the node that we would want to remove, so no additional step is needed.

For the second removal, we now move 13 (the node that we'd have to remove to maintain a complete tree) to the position below (right of) 15 (see second figure above, in part b). This requires swapping up once (after that, 7 is the parent, and $7 < 13$ so we're done):



Now, the second method (the more efficient one, since it does a single traversal of a path of length h, instead of two, like the other method — even though as we know, for the average case, the adjustment upwards has an average run time of $\Theta(1)$, so the difference is not that huge after all) consists of moving the last node to the root, and then percolate it down (always swapping with the lower of the two children — otherwise, we would break the Heap structure for the other sub-tree).

Removing 7 then causes 15 to move to the root, and then swap with 9 (the lower of the two children):



(etc.)

4 – Show the results for question 3 using an array representation for the heap (size and resizing policy is not really relevant for this exercise — you may assume a fixed-size array of size 8).

Solution:

The only key detail to keep in mind here is the formulas to obtain children nodes and parent node (necessary for percolations down and percolations up, respectively).

Leaving position 0 of the array unused and assigning the root node at position 1, we have that for node at position k, its children are at positions $2k$ and $2k + 1$, and its parent (if $k > 1$) is at position $k \div 2$ (integer division).

5 – (If enough time) A more or less optimized form of the bubble-sort makes the passes alternating direction — instead of doing n times the same loop, from 0 to $n - 1$ (or rather, from 0 to $n - i - 1$, where i is the outer loop's counter) always, we do it one time ascending, then one time descending, then ascending, and so on.

Another typical optimization uses the fact that we can stop after an outer loop's iteration where there were no swaps after completing the inner loop.

Write a C++ fragment of code to implement this optimized version of bubble-sort.

Solution:

Couple of details: we keep a boolean flag for the swaps; we set it to false at the beginning of each outer loop's pass, and set it to true whenever there's a swap — the outer loop stops when that flag is false.

We also keep a "step" variable, containing either $+1$ or -1, to alternate directions. We also should keep in mind that after iteration 1 (outer loop), the largest value is at the end. After iteration 2, the lowest value is at the beginning, so the outer loop should be adjusted according to these:

```
template <typename Type>
void bubble_sort (Type * array, int n)
{
    bool swaps = true;
    int start = 0, end = n-1;
    int dir = 1;  // direction --- alternate +1 / -1
    int i = start;
    while (swaps)
    {
        swaps = false;
        for ( ; i != (dir > 0 ? end : start); i+= dir)
        {
            if (array[std::min(i,i+dir)] > array[std::max(i,i+dir)])
            {
                swaps = true;
                std::swap (array[i], array[i+dir]);
                    // why did we need std::min and max in the if
                    // but not here?
            }
        }

        if (dir > 0)
        {
            --end;
        }
        else
        {
            ++start;
        }
        dir = -dir;
    }
}
```