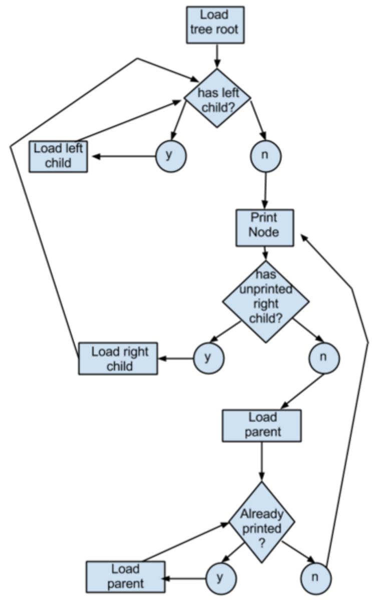
Tree sort

A **tree sort** is a sort algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree (in-order) so that the elements come out in sorted order. Its typical use is sorting elements online: after each insertion, the set of elements seen so far is available in sorted order.

Tree sort



Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ (unbalanced) $O(n \log n)$ (balanced)
Best-case performance	$O(n \log n)$
Average performance	$O(n \log n)$
Worst-case space complexity	$\Theta(n)$

Efficiency

Adding one item to a binary search tree is on average an $O(\log n)$ process (in big O notation). Adding n items is an $O(n \log n)$ process, making tree sorting a 'fast sort' process. Adding an item to an unbalanced binary tree requires $O(n)$ time in the worst-case: When the tree resembles a linked list (degenerate tree). This results in a worst case of $O(n^2)$ time for this sorting algorithm. This worst case occurs when the algorithm operates on an already sorted set, or one that is nearly sorted. Expected $O(n \log n)$ time can however be achieved by shuffling the array.

The worst-case behaviour can be improved by using a self-balancing binary search tree. Using such a tree, the algorithm has an $O(n \log n)$ worst-case performance, thus being degree-optimal for a comparison sort. However, trees require memory to be allocated on the heap, which is a significant performance hit when compared to quicksort and heapsort. When using a splay tree as the binary search tree, the resulting algorithm (called splaysort) has the additional property that it is an adaptive sort, meaning that its running time is faster than $O(n \log n)$ for inputs that are nearly sorted.

Example

The following tree sort algorithm in pseudocode accepts a collection of comparable items and outputs the items in ascending order:

```
STRUCTURE BinaryTree
    BinaryTree:LeftSubTree
    Object:Node
    BinaryTree:RightSubTree

PROCEDURE Insert(BinaryTree:searchTree, Object:item)
    IF searchTree.Node IS NULL THEN
        SET searchTree.Node TO item
    ELSE
        IF item IS LESS THAN searchTree.Node THEN
            Insert(searchTree.LeftSubTree, item)
        ELSE
            Insert(searchTree.RightSubTree, item)

PROCEDURE InOrder(BinaryTree:searchTree)
    IF searchTree.Node IS NULL THEN
        EXIT PROCEDURE
    ELSE
        InOrder(searchTree.LeftSubTree)
        EMIT searchTree.Node
        InOrder(searchTree.RightSubTree)

PROCEDURE TreeSort(Collection:items)
```

```
    BinaryTree:searchTree

    FOR EACH individualItem IN items
        Insert(searchTree, individualItem)

    InOrder(searchTree)
```

In a simple functional programming form, the algorithm (in Haskell) would look something like this:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)

insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node Leaf x Leaf
insert x (Node t y s)
    | x <= y = Node (insert x t) y s
    | x > y  = Node t y (insert x s)

flatten :: Tree a -> [a]
flatten Leaf = []
flatten (Node t x s) = flatten t ++ [x] ++ flatten s

treesort :: Ord a => [a] -> [a]
treesort = flatten . foldr insert Leaf
```

In the above implementation, both the insertion algorithm and the retrieval algorithm have $O(n^2)$ worst-case scenarios.

External links

- Binary Tree Java Applet and Explanation (http://www.qmatica.com/DataStructures/Trees/BST.html)
- Tree Sort of a Linked List (http://www.martinbroadhurst.com/articles/sorting-a-linked-list-by-turning-it-into-a-binary-tree.html)
- Tree Sort in C++ (http://www.martinbroadhurst.com/cpp-sorting.html#tree-sort)