

Binary Tree Search

For searching a binary tree we assume that each node has a record with a key value. We also insist that all records with smaller keys are placed in the left subtree and that all records with a larger key value are placed in the right subtree. For example the data A S E A R C H will be arranged in a tree as follows:

An algorithm for searching such a tree might be:

```
static struct node
{ int key, info; struct node *l , *r};
static struct node *t, *head, *z;

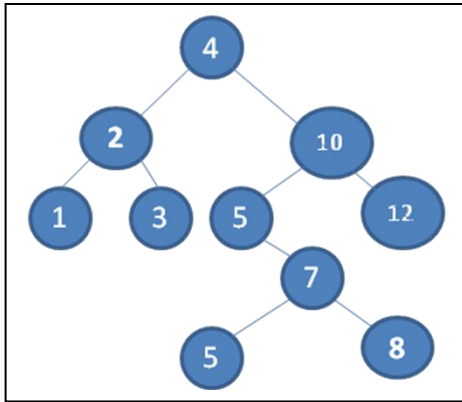
int treesearch(int v)
{
    struct node *x = head->r;
    z->key = v;
    while (v != x->key)
        if (v < x->key) then
            x = x->l;
        else
            x = x->r;

    return x-> info;
}
```

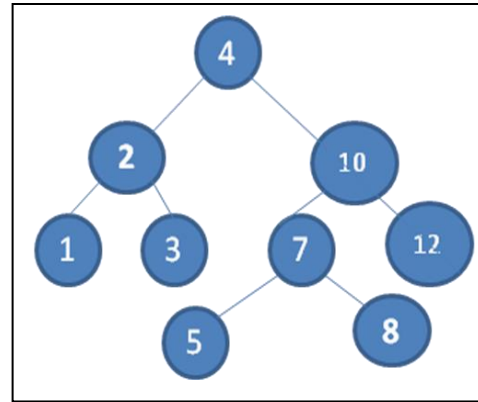
Note we assume that an empty node points to a tail node called z. Initializing this with the search data ensures we will always complete the search, we can know it's unsuccessful by placing -1 in **info**.

Deletion

To delete a single leaf node simply remove it from the tree. This is trivial and gets only a little more complicated when we delete a node with one child. Here we simply remove the node and replace it with the child as shown in the following example.

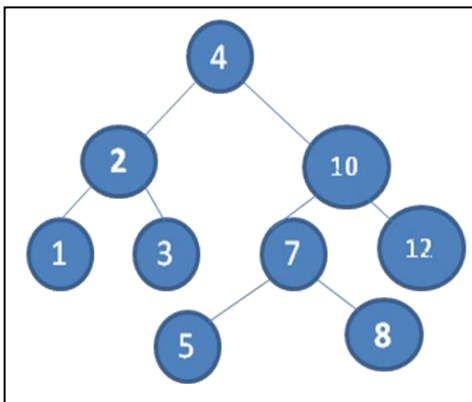


Before

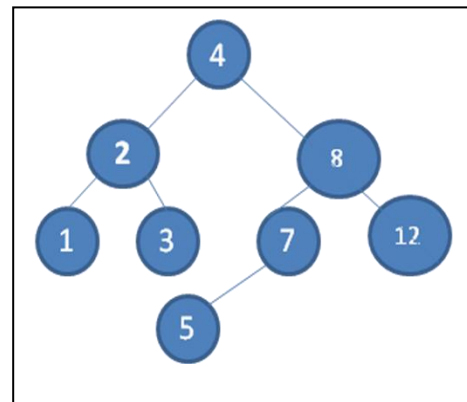


After

To delete a node with 2 children is a little more tricky. Remember that an in-order traversal will result in a sorted list. Using this fact we choose the in-order predecessor node of the node to be deleted and swap the values in these nodes. Then we delete the predecessor node. Alternatively we could choose the in-order successor node and repeat the process with this node. Either will work. See the following example where we delete the node containing the number 10.



Before



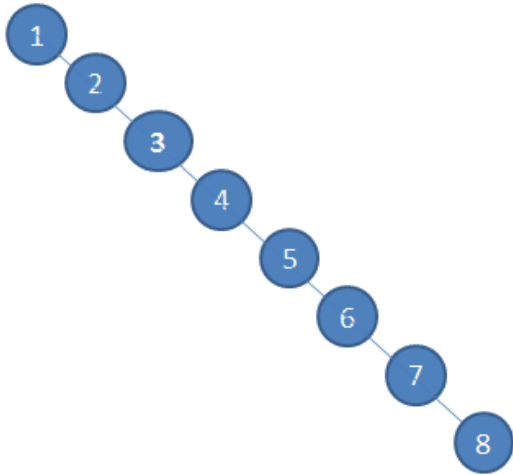
After

Complexity.

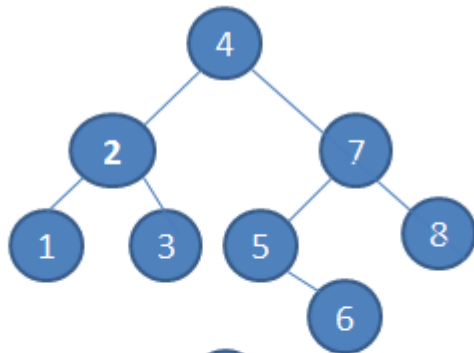
Since each step of a binary search tree algorithm will remove half the data, then searching for an item has complexity of $O(\log(n))$ if the tree is balanced (all left and right subtrees are equally populated). Once we have found an item to delete the actual deletion takes a constant time, hence overall a deletion has $O(\log(n))$ complexity. Similarly insertion is also $O(\log(n))$ in its complexity.

Over time insertions and deletions in a binary search tree like this will ultimately lead to a very unbalanced tree and the resulting performance of the search will degenerate to $O(n)$ assuming the item to be search for

is always a leaf. We must keep the tree balanced to avoid this. The following example demonstrated how insertion of pre-sorted data into the search tree will result in a tree so unbalanced that it becomes a list.



Various algorithms exist that will keep a tree balanced and we will meet these later. For now just note that for a given set of data there are many ways to store the data as a balance tree. Here are two examples showing this. The order in which the data is added to the tree will determine how the tree evolves.



Or:

