Binary search trees

Binary search trees :

Search trees are data structures that generally offer the following dynamic-set operations :

- SEARCH
- MINIMUM
- MAXIMUM
- PREDECESSOR
- SUCCESSOR
- INSERT
- DELETE

Basic operations on these trees take time proportional to the height of the tree. For complete balanced trees with n nodes, this height is of `log` n. However, since trees are not always balanced, the worst case is in $\theta(n)$.

To represent the nodes of these tree, we usually use linked data structures in which each node of the tree is an object. These have the following fields :

- *key*: The usual value used to compare the different objects.
- *p* : The parent of the current node.
- *left* : The left child of the current node.
- *right* : The right child of the current node.

Binary-search-tree property :

> Let x be a node in a binary search tree.
> - If y is a node in the left subtree of x, then $key[y] \leq key[x]$.
> - If y is a node in the right subtree of x, then $key[x] \leq key[y]$.

This property allows us to easily output all the keys in a binary search tree in sorted order :

```
function INORDER − TREE − WALK(x)
    if x ≠ NIL
        then
            INORDER − TREE − WALK(left[x])
            print key[x]
            INORDER − TREE − WALK(right[x])
```

Note that the name *inorder tree walk* comes from the fact that this algorithm prints the value of the key of the current node between the values of the left subtree and the values of the right subtree.

Similarly, a *preorder tree walk* prints the root before the values in either subtree, and a *postorder tree walk* prints the root after the values in its subtrees.

Searching :

```
function TREE − SEARCH(x, k)
    if x = NIL or k = key[x]
        then return x
    if k < key[x]
        then return TREE − SEARCH(left[x], k)
        else return TREE − SEARCH(right[x], k)
```

This algorithm starts at node x and traces its way downward until it finds the node with the key equal to k or determines that no such node exist.

The maximum number of keys encountered during the recursive search is the length of the path going from x to the node with key k. The running time is therefore in $O(h)$, where h is the height of the tree.

```
function ITERATIVE − TREE − SEARCH(x, k)
    while x ≠ NIL and k ≠ key[x]
        do if k < key[x]
            then x ← left[x]
            else x ← right[x]
```

Minimum and Maximum :

The minimum in a binary-search-tree can always be found by following the *left* pointers from the root until a NIL is encountered.

```
function TREE − MINIMUM(x)
    while left[x] ≠ NIL
        do x ← left[x]
    return x
```

Similarly, the maximum can be found by following the *right* pointers until a NIL is found.

```
function TREE − MAXIMUM(x)
    while right[x] ≠ NIL
        do x ← right[x]
    return x
```

Both of these algorithms are in $O(h)$ since they trace paths downward in the tree.

Predecessor and Successor :

If all keys are distinct, the successor of a node x is the node with the smallest key greater then $key[x]$. The following algorithm returns this successor or NIL if the key of x is the largest key in the tree.

```
function TREE − SUCCESSOR(x)
    if right[x] ≠ NIL
        then return TREE − MINIMUM(right[x])
    y ← p[x]
    while y ≠ NIL and x = right[y] do
        x ← y
        y ← p[y]
    return y
```

If the right subtree of x is not NIL, then the sucessor is the minimum of this subtree. Otherwise, the returned value is the closest ancestor who's left subtree includes x.

The algorithm for $TREE-PREDECESSOR$ is the symmetry of the $TREE-SUCCESSOR$ algorithm.

The running time for a tree of height h is $O(h)$.

Insertion :

```
function TREE − INSERT(T, z)
    y ← NIL
    x ← root[T]
    while x ≠ NIL do
        y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
    p[z] ← y
    if y = NIL
        then root[T] ← z
        else
            if key[z] < key[y]
                then left[y] ← z
                else right[y] ← z
```

This algorithm starts at the *root* of the tree and traces a path downward. The pointer x traces this path while the pointer y is maintained as the parent of x. When x is found to be NIL then the appropriate position has been found and the new value can be inserted as a child of y.

Like the previous algorithms, $TREE - INSERT$ is in $O(h)$.

Deletion :

```
function TREE − DELETE(T, z)
    if left[z] = NIL or right[z] = NIL
        then y ← z
        else y ← TREE − SUCCESSOR(z)
    if left[y] ≠ NIL
        then x ← left[y]
        else x ← right[y]
    if x ≠ NIL
        then p[x] ← p[y]
    if p[y] = NIL
        then root[T] ← x
        else
            if y = left[p[y]]
                then left[p[y]] ← x
                else right[p[y]] ← x
    if y ≠ z
        then key[z] ← key[y]
        \\ If y has other fields, copy them, too.
    return y
```

This algorithm, like all others, is in $O(h)$.